SCIENTIFIC COMPUTING BY NUMERICAL METHODS

Christina C. Christara and Kenneth R. Jackson,

Computer Science Dept., University of Toronto, Toronto, Ontario, Canada, M5S 1A4.

(ccc@cs.toronto.edu and krj@cs.toronto.edu)

Contents

Introduction

Numerical methods are an indispensable tool in solving many problems that arise in science and engineering. In this article, we briefly survey a few of the most common mathematical problems and review some numerical methods to solve them.

As can be seen from the table of contents, the topics covered in this survey are those that appear in most introductory numerical methods books. However, our discussion of each topic is more brief than is normally the case in such texts and we frequently provide references to more advanced topics that are not usually considered in introductory books.

Definitions of some common mathematical terms used in this survey can be found in the glossary at the end of the article. We also list some common mathematical symbols and abbreviations used throughout the survey in the two sections following the glossary.

1 Floating-Point Arithmetic

In this section, we consider the representation of floating-point numbers, floating-point arithmetic, rounding errors, and the effects of inexact arithmetic in some simple examples. For a more detailed discussion of these topics, see (Wilkinson 1965; Goldberg 1991) or an introductory numerical methods text.

1.1 The IEEE Standard

The approval of the IEEE[1] Standard for Binary Floating-Point Arithmetic (IEEE 1985) was a significant advance for scientific computation. Not only has this led to cleaner floating-point arithmetic than was commonly available previously, thus greatly facilitating the development of reliable, robust numerical software, but, because many computer manufacturers have since adopted the standard, it has significantly increased the portability of programs.

The IEEE standard specifies both single- and double-precision floating-point numbers, each of the form

$$(-1)^s \times b_0 \,.\, b_1 \, b_2 \, \cdots \, b_{p-1} \times 2^E \tag{1}$$

where $s = 0$ or 1, $(-1)^s$ is the *sign* of the number, $b_i = 0$ or 1 for $i = 0, \ldots, p-1$, $b_0 \,.\, b_1 \, b_2 \, \cdots \, b_{p-1}$ is the *significand* (sometimes called the *mantissa*) of the number, and the *exponent* E is an integer satisfying $E_{\min} \leq E \leq E_{\max}$. In single-precision, $p = 24$, $E_{\min} = -126$ and $E_{\max} = +127$, whereas, in double-precision, $p = 53$, $E_{\min} = -1022$ and $E_{\max} = +1023$. We emphasize that a number written in the form (1) is binary. So, for example, $1.100 \cdots 0 \times 2^0$ written in the format (1) is equal to the decimal number 1.5.

A *normalized* number is either 0 or a floating-point number of the form (1) with $b_0 = 1$ (and so it is not necessary to store the leading bit). In single-precision, this provides the equivalent of 7 to 8 significant decimal digits with positive and negative numbers having magnitudes roughly in the range $[1.2 \times 10^{-38}, 3.4 \times$

[1] Institute of Electrical and Electronics Engineers

$10^{+38}]$. In double-precision, this is increased to about 16 significant decimal digits and a range of roughly $[2.2 \times 10^{-308}, 1.8 \times 10^{+308}]$.

An *underflow* occurs when an operation produces a nonzero result in the range $(-2^{E_{\min}}, +2^{E_{\min}})$. In the IEEE standard, the default is to raise an underflow exception flag and to continue the computation with the result correctly rounded to the nearest *denormalized* number or zero. A denormalized floating-point number has the form (1) with $E = E_{\min}$ and $b_0 = 0$. Because denormalized numbers use some of the leading digits from the significand to represent the magnitude of the number, there are fewer digits available to represent its significant digits. Using denormalized numbers in this way for underflows is sometimes referred to as *gradual underflow*. Many older non-IEEE machines do not have denormalized numbers and, when an underflow occurs, they either abort the computation or replace the result by zero.

An *overflow* occurs when an operation produces a nonzero result outside the range of floating-point numbers of the form (1). In the IEEE standard, the default is to raise an overflow exception flag and to continue the computation with the result replaced by either $+\infty$ or $-\infty$, depending on the sign of the overflow value. Many older non-IEEE machines do not have $+\infty$ or $-\infty$ and, when an overflow occurs, they usually abort the computation.

The IEEE standard also includes at least two NaNs (Not-a-Number) in both precisions, representing indeterminate values that may arise from invalid or inexact operations such as $(+\infty) + (-\infty)$, $0 \times \infty$, $0/0$, ∞/∞ or $\sqrt{x}$ for $x < 0$. When a NaN arises in this way, the default is to raise an exception flag and to continue the computation. This novel feature is not available on most older non-IEEE machines.

It follows immediately from the format (1) that floating-point numbers are discrete and finite, while real numbers are dense and infinite. As a result, an arithmetic operation performed on two floating-point numbers may return a result that cannot be represented exactly in the form (1) in the same precision as the operands.

A key feature of the IEEE standard is that it requires that the basic arithmetic operations $+$, $-$, $\times$, $/$ and $\sqrt{\ }$ return *properly rounded* results. That is, we may think of the operation as first being done exactly and then properly rounded to the precision of the result. An operation with $\pm\infty$ is interpreted as the limiting case of the operation with an arbitrary large value in place of the ∞, when such an interpretation makes sense; otherwise the result is a NaN. An operation involving one or more NaNs returns a NaN.

The default rounding mode is *round-to-nearest*: that is, the exact result of the arithmetic operation is rounded to the nearest floating-point number, where, in the case of a tie, the floating-point number with the least significant bit equal to 0 is selected. The standard also provides for directed roundings (round-towards-$+\infty$, round-towards-$-\infty$ and round-towards-0), but these are not easily accessed from most programming languages.

Another important feature of the IEEE standard is that comparisons are exact and never overflow or underflow. The comparisons $<$, $>$ and $=$ work as expected with finite floating-point numbers of the form (1) and $-\infty < x < +\infty$ for any finite floating-point number x. NaNs are unordered and the comparison of a NaN with any other value — including itself — returns false.

The IEEE standard also provides for *extended* single- and double-precision floating-point numbers, but

these are not easily accessed from most programming languages, so we do not discuss then here.

As noted above, the IEEE standard has been widely adopted in the computer industry, but there are several important classes of machines that do not conform to it, including Crays, DEC Vaxes and IBM mainframes. Although their floating-point numbers are similar to those described above, there are important differences. Space limitations, though, do not permit us to explore these systems here.

1.2 Rounding Errors

Since IEEE standard floating-point arithmetic returns the correctly rounded result for the basic operations $+$, $-$, $\times$, $/$ and $\sqrt{\ }$, one might expect that rounding errors would never pose a problem, particularly in double-precision computations. Although rounding errors can be ignored in many cases, the examples in the next subsection show that they may be significant even in simple calculations. Before considering these examples, though, we need to define an important machine constant and describe its significance in floating-point computation.

Machine epsilon, often abbreviated *mach-eps*, is the distance from 1 to the next larger floating-point number. For numbers of the form (1), mach-eps $= 2^{1-p}$. So for IEEE single- and double-precision numbers, mach-eps is $2^{-23} \approx 1.19209 \times 10^{-7}$ and $2^{-52} \approx 2.22045 \times 10^{-16}$, respectively.

A common alternate definition of machine epsilon is that it is the smallest positive floating-point number ϵ such that $1 + \epsilon > 1$ in floating-point arithmetic. We prefer the definition given in the previous paragraph, and use it throughout this section, because it is independent of the rounding mode and so is characteristic of the floating-point number system itself, while the alternate definition given in this paragraph depends on the rounding mode as well. We also assume throughout this section that round-to-nearest is in effect. The discussion below, though, can be modified easily for the alternate definition of mach-eps and other rounding modes.

It follows immediately from the floating-point format (1) that the absolute distance between floating-point numbers is not uniform. Rather, from (1) and the definition of mach-eps above, we see that the spacing between floating-point numbers in the intervals $[2^k, 2^{k+1})$ and $(-2^{k+1}, -2^k]$ is mach-eps $\times\, 2^k$ for $E_{\min} \le k \le E_{\max}$. Thus, the absolute distance between neighbouring nonzero normalized floating-point numbers with the same exponent is uniform, but the floating-point numbers near $2^{E_{\min}}$ are much closer together in an absolute sense than those near $2^{E_{\max}}$. However, the *relative* spacing between all nonzero normalized floating-point numbers does not vary significantly. It is easy to see that, if x_1 and x_2 are any two neighbouring nonzero normalized floating-point numbers, then

$$\frac{\text{mach-eps}}{2} \le \left| \frac{x_1 - x_2}{x_1} \right| \le \text{mach-eps}. \tag{2}$$

As a result, it is more natural to consider relative, rather than absolute, errors in arithmetic operations on floating-point numbers, as is explained in more detail below.

For $x \in \mathbb{R}$, let $fl(x)$ be the floating-point number nearest to x, where, in the case of a tie, the floating-point number with the least significant bit equal to 0 is selected. The importance of mach-eps stems largely

from the observation that, if $fl(x)$ does not overflow or underflow, then

$$fl(x) = x(1+\delta) \text{ for some } |\delta| \leq u, \tag{3}$$

where u, the *relative roundoff error bound*, satisfies $u =$ mach-eps/2 for round-to-nearest and $u =$ mach-eps for the other IEEE rounding modes. Rewriting (3) as

$$\delta = \frac{fl(x) - x}{x}$$

we see that δ is the relative error incurred in approximating x by $fl(x)$, and so (3) is closely related to (2).

If op is one of $+$, $-$, $\times$ or $/$ and x and y are two floating-point numbers, let $fl(x \text{ op } y)$ stand for the result of performing the arithmetic operation x op y in floating-point arithmetic. If no arithmetic exception arises in the floating-point operation, then

$$fl(x \text{ op } y) = (x \text{ op } y)(1+\delta) \text{ for some } |\delta| \leq u, \tag{4}$$

where the $(x \text{ op } y)$ on the right side of (4) is the exact result of the arithmetic operation. Similarly, if x is a nonnegative normalized floating-point number, then

$$fl(\sqrt{x}) = \sqrt{x}\,(1+\delta) \text{ for some } |\delta| \leq u. \tag{5}$$

Again, the u in either (4) or (5) is the relative roundoff error bound and the δ is the relative error incurred in approximating x op y by $fl(x \text{ op } y)$ or $\sqrt{x}$ by $fl(\sqrt{x})$, respectively.

Although the relations (4)–(5) are not quite as tight as the requirement that the basic operations $+$, $-$, $\times$, $/$ and $\sqrt{\ }$ return the correctly rounded result, they are very useful in deriving error bounds and explaining the effects of rounding errors in computations.

1.3 The Effects of Inexact Arithmetic: Some Illustrative Examples

As noted at the start of the last subsection, since IEEE standard floating-point arithmetic returns the correctly rounded result for the basic operations $+$, $-$, $\times$, $/$ and $\sqrt{\ }$, one might expect that rounding errors would never pose a problem, particularly in double-precision computations. This, though, is not the case. In this subsection we consider a simple example that illustrates some of the *pitfalls* of numerical computation.

Suppose that we compute the expression

$$1 + 10^{10} - 10^{10} \tag{6}$$

in single-precision from left to right. We first compute $fl(1 + 10^{10}) = 10^{10}$, the correctly rounded single-precision result. Then we use this value to compute $fl(10^{10} - 10^{10}) = 0$, without committing an additional rounding error. Thus, $fl(\,(1 + 10^{10}) - 10^{10}) = 0$, whereas the true result is 1.

The key point to note here is that $fl(1 + 10^{10}) = 10^{10} = (1 + 10^{10})(1 + \delta)$, where $|\delta| = 1/(1 + 10^{10}) <$

$10^{-10} < u = 2^{-24}$. So the rounding error that we commit in computing $1 + 10^{10}$ is small relative to $1 + 10^{10}$, the true result of the first addition, but the absolute error of 1 associated with this addition is not small compared to the true final answer, 1, thus illustrating the **Rule for Sums**:

> *Although a rounding error is always small relative to the result that gives rise to it, it might be large relative to the true final answer if intermediate terms in a sum are large relative to the true final answer.*

The Rule for Sums is important to remember when computing more complex expressions such as the truncated Taylor series $T_k(x) = 1 + x + x^2/2 + \cdots + x^k/k! \approx e^x$, where k is chosen large enough so that the *truncation error*

$$e^x - T_k(x) = \sum_{i=k+1}^{\infty} x^i/i!$$

is insignificant relative to e^x. It is easy to prove that, if $x \geq 0$ and k is large enough, then $T_k(x)$ is a good approximation to e^x. However, if $x < 0$ and of moderate magnitude, then the rounding error associated with some of the intermediate terms $x^i/i!$ in $T_k(x)$ might be much larger in magnitude than either the true value of $T_k(x)$ or e^x. As a result, $fl(T_k(x))$, the computed value of $T_k(x)$, might be completely erroneous, no matter how large we choose k. For example, $e^{-15} \approx 3.05902 \times 10^{-7}$, while we computed $fl(T_k(x)) \approx 2.12335 \times 10^{-2}$ in IEEE single-precision arithmetic on a Sun Sparcstation.

A similar problem is less likely to occur with multiplications, provided no overflow or underflow occurs, since from (4)

$$fl(x_1 \cdot x_2 \cdots x_n) = x_1 \cdot x_2 \cdots x_n(1 + \delta_1) \cdots (1 + \delta_{n-1}) \tag{7}$$

where $|\delta_i| \leq u$ for $i = 1, \ldots, n-1$. Moreover, if $nu \leq 0.1$, then $(1 + \delta_1) \cdots (1 + \delta_{n-1}) = 1 + 1.1\, n\, \delta$ for some $\delta \in [-u, u]$. Therefore, unless n is very large, $fl(x_1 \cdot x_2 \cdots x_n)$ is guaranteed to be a good approximation to $x_1 \cdot x_2 \cdots x_n$. However, it is not hard to find examples for which $nu \gg 1$ and $fl(x_1 \cdot x_2 \cdots x_n)$ is a poor approximation to $x_1 \cdot x_2 \cdots x_n$.

The example (6) also illustrates another important phenomenon commonly called *catastrophic cancelation*: all the digits in the second sum, $fl(10^{10} - 10^{10})$, cancel, signaling a catastrophic loss of precision. Catastrophic cancelation refers also to the case that many, but not all, of the digits cancel. This is often a sign that a disastrous loss of accuracy has occurred, but, as in this example when we compute $fl(1+10^{10}) = 10^{10}$ and loose the 1, it is often the case that the accuracy is lost before the catastrophic cancelation occurs.

For an example of catastrophic cancelation in a more realistic computation, consider calculating the roots of the quadratic $ax^2 + bx + c$ by the standard formula

$$r_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{8}$$

We used this formula to compute the roots of the quadratic $x^2 - 10^4 x + 1$ in IEEE single-precision arithmetic on a Sun Sparcstation. The computed roots were 10^4 and 0, the larger of which is accurate, having a relative error of about 10^{-8}, but the smaller one is completely wrong, the true root being about 10^{-4}. A similar result usually occurs whenever $|ac|/b^2 \ll 1$. The root of larger magnitude is usually computed precisely,

but the smaller one is frequently very inaccurate due to catastrophic cancelation, since $b^2 - 4ac \approx b^2$ and so $|b| - \sqrt{b^2 - 4ac} \approx 0$. Although the second of these relations signals catastrophic cancelation, the loss of precision occurs in the first.

There is an easy remedy for the loss of precision due to catastrophic cancelation in this case. Use (8) to compute, r_1, the root of larger magnitude, and then use the alternate formula $r_2 = c/(a \cdot r_1)$ to compute the smaller one. The relative error in r_2 is at most $(1+u)^2$ times larger than the relative error in r_1, provided that no overflows or underflows occur in computing $c/(a \cdot r_1)$.

Another point to note is that, if we compute (6) from right to left, instead of left to right, then

$$fl(1 + (10^{10} - 10^{10})) = fl(1 + 0) = 1.$$

It is not particularly significant that this computation gives the correct answer, but what is important is that it illustrates that floating-point addition is not associative, although it is commutative, since $fl(a+b)$ and $fl(b+a)$ are both required to be the correctly rounded value for $a + b = b + a$. Similar results hold for multiplication and division.

Many other fundamental mathematical relations that we take for granted do not hold for floating-point computations. For example, the result that $\sin(x)$ is strictly increasing for $x \in (0, \pi/2)$ cannot hold in any floating-point system in which x and $\sin(x)$ are in the same precision, since there are more floating-point numbers in the domain $(0, \pi/2)$ than there are in $(0, 1)$, the range of $\sin(x)$.

Finally, we end this section by noting that overflows and underflows often cause problems in computations. After an overflow, $\pm\infty$ or NaN frequently propagates through the computation. Although this can sometimes yield a useful result, it is more often a signal of an error in the program or its input. On the other hand, continuing the computation with denormalized numbers of zero in place of an underflow can often yield a useful numerical result. However, there are cases when this can be disastrous. For example, if x^2 underflows, but y^2 is not too close to the underflow limit, then $fl(\sqrt{x^2 + y^2})$, the computed value of $\sqrt{x^2 + y^2}$, is still accurate. However, if both x^2 and y^2 underflow to 0, then $fl(\sqrt{x^2 + y^2}) = 0$, although $\sqrt{x^2 + y^2} \geq \max(|x|, |y|)$ may be far from the underflow limit.

It is often possible to ensure that overflows do not occur and that underflows are harmless. For the example considered above, note that $\sqrt{x^2 + y^2} = s\sqrt{(x/s)^2 + (y/s)^2}$ for any scaling factor $s > 0$. If we choose $s = 2^k$ for an integer $k \approx \log_2(\max(|x|, |y|))$, then neither $(x/s)^2$ nor $(y/s)^2$ can overflow, and any underflow that occurs is harmless, since one of $(x/s)^2$ or $(y/s)^2$ is close to 1. Moreover, in IEEE floating-point arithmetic, multiplying and dividing by $s = 2^k$ does not introduce any additional rounding error into the computation.

A similar problem with overflows and underflows occurs in formula (8) and in many other numerical computations. Overflows can be avoided and underflows can be rendered harmless in computing $\sqrt{b^2 - 4ac}$ in (8) by scaling in much the same way as described above for $\sqrt{x^2 + y^2}$.

2 The Direct Solution of Linear Algebraic Systems

In this section, we consider the *direct* solution of linear algebraic systems of the form $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ (or $\mathbb{C}^{n \times n}$) is a nonsingular matrix and x and $b \in \mathbb{R}^n$ (or $\mathbb{C}^n$). A numerical method for solving $Ax = b$ is direct if it computes the exact solution of the system when implemented in exact arithmetic. Iterative methods for nonsingular linear systems and methods for over-determined and under-determined systems are considered in §4 and §5, respectively.

The standard direct methods for solving $Ax = b$ are based on, or closely related to, *Gaussian elimination* (GE), the familiar variable elimination technique that reduces the original system $Ax = b$ to an *upper-triangular system*, $Ux = \tilde{b}$, which has the same solution x. We present a simple form of GE in §2.1 and show how $Ux = \tilde{b}$ can be solved easily by *back substitution* in §2.2. We then explain how the simple form of GE presented in §2.1 for $Ax = b$ relates to the LU factorization of the coefficient matrix A in §2.3 and to *forward elimination* in §2.4. Enhancements to this simple form of GE to make it an efficient, robust, reliable numerical method for the solution of linear systems are outlined in §2.5. The closely related Cholesky factorization for symmetric positive-definite matrices is presented in §2.6. We consider how to adapt these methods to banded and sparse linear systems in §2.7. We end with a discussion of the effects of rounding errors on the direct methods in §2.8 and of *iterative improvement*, a technique to ameliorate these effects, in §2.9. See §13 for a discussion of sources of high-quality numerical software for solving systems of linear algebraic equations.

GE can also be applied to singular systems of linear equations or over-determined or under-determined linear systems of m equations in n unknowns. However, it is not as robust as the methods discussed in §5 for these problems, so we do not present these generalizations of GE here. In addition, we note that there are several mathematically equivalent, but computationally distinct, implementations of GE and the factorizations discussed here. The reader interested in a more comprehensive treatment of these topics should consult an advanced text, such as Golub and Van Loan (1989).

Finally we note that it is generally inadvisable to solve a system $Ax = b$ by first computing A^{-1} and then calculating $x = A^{-1}b$. The techniques discussed in this section are usually both more reliable and more cost effective than methods using A^{-1}. We also note that, although Cramer's rule is a useful theoretical tool, it is an extremely ineffective computational scheme.

2.1 Gaussian Elimination

First we establish some notation used throughout this section. The linear algebraic system to be solved is $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ (or $\mathbb{C}^{n \times n}$) is a nonsingular matrix and x and $b \in \mathbb{R}^n$ (or $\mathbb{C}^n$) are vectors. To unify the notation used below, let $A_0 = A$ and $b_0 = b$.

Gaussian elimination (GE) for $Ax = b$ proceeds in $n-1$ stages. For $k = 1, \ldots, n-1$, we begin stage k of GE with the reduced system $A_{k-1}x = b_{k-1}$, where columns $1, \ldots, k-1$ of A_{k-1} contain 0's below the main diagonal. That is, for $A_{k-1} = [a_{ij}^{(k-1)}]$, $a_{ij}^{(k-1)} = 0$ for $j = 1, \ldots, k-1$ and $i = j+1, \ldots, n$. This corresponds to the variables $x_1, \ldots, x_{i-1}$ having been eliminated from equation i of $A_{k-1}x = b_{k-1}$ for $i = 2, \ldots, k-1$

and the variables $x_1, \ldots, x_{k-1}$ having been eliminated from the remaining equations $k, \ldots, n$. Moreover, x is the unique solution of both $A_{k-1}x = b_{k-1}$ and $Ax = b$. Note that all the assumptions above hold vacuously for $k = 1$, since, in this case, the "reduced" system $A_0 x = b_0$ is just the original system $Ax = b$ from which no variables have yet been eliminated.

During stage k of GE, we further the reduction process by eliminating the variable x_k from row i of $A_{k-1}x = b_{k-1}$ for $i = k+1, \ldots, n$ by multiplying row k of this system by $m_{ik} = a_{ik}^{(k-1)}/a_{kk}^{(k-1)}$ and subtracting it from row i.

Note that the multipliers m_{ik} are not properly defined by $m_{ik} = a_{ik}^{(k-1)}/a_{kk}^{(k-1)}$ if the *pivot element* $a_{kk}^{(k-1)} = 0$. In exact arithmetic, this can happen only if the $k \times k$ leading principal minor of A is singular. We consider how to deal with zero (or nearly zero) pivots in §2.5. For now, though, we say that this simple form of GE "breaks down" at stage k and we terminate the process.

After the last stage of GE, the original system $Ax = b$ has been reduced to $Ux = \tilde{b}$, where $\tilde{b} = b_{n-1}$ and $U = A_{n-1}$. Note that $U = [u_{ij}]$ is an upper-triangular matrix (i.e., $u_{ij} = 0$ for $1 \le j < i \le n$) with $u_{kk} = a_{kk}^{(k-1)}$ for $k = 1, \ldots, n$. Therefore, if A and all its leading principal minors are nonsingular, then $u_{kk} = a_{kk}^{(k-1)} \neq 0$ for $k = 1, \ldots, n$, so U is nonsingular too. Moreover, in this case, x is the unique solution of both $Ax = b$ and $Ux = \tilde{b}$. The latter system can be solved easily by back substitution, as described in §2.2.

The complete GE process can be written in pseudo-code as shown in Table 1. Note that at stage k of GE, we know the elements $a_{ik}^{(k)} = a_{ik}^{(k-1)} - m_{ik} \cdot a_{kk}^{(k-1)} = 0$ for $i = k+1, \ldots, n$, so we do not need to perform this calculation explicitly. Consequently, instead of j running for $k, \ldots, n$ in Table 1, as might be expected, j runs from $k+1, \ldots, n$ instead.

To reduce the storage needed for GE, the original matrix A and vector b are often overwritten by the intermediate matrices A_k and vectors b_k, so that at the end of the GE process, the upper-triangular part of A contains U and b contains $\tilde{b}$. The only change required to the algorithm in Table 1 to implement this reduced storage scheme is to remove the superscripts from the coefficients a_{ij} and b_i. As explained below, it is also important to store the multipliers m_{ik}. Fortunately, the $n-k$ multipliers $\{m_{ik} : i = k+1, \ldots, n\}$, created at stage k of GE can be stored in the $n-k$ positions in column k of A below the main diagonal that are eliminated in stage k. Thus, in many implementations of GE, the upper-triangular part of A is overwritten by U and the strictly lower part of A is overwritten by the multipliers m_{ik} for $1 \le k < i \le n$.

A straightforward count of the operations in Table 1 shows that GE requires $n(n-1)/2 \approx n^2/2$ divisions to compute the multipliers m_{ik}, $n(2n-1)(n-1)/6 \approx n^3/3$ multiplications and subtractions to compute the coefficients of U, and $n(n-1)/2 \approx n^2/2$ multiplications and subtractions to compute the coefficients of $\tilde{b}$. Since multiplications and subtractions (or additions) occur in pairs so frequently in matrix calculations, we refer to this pair of operations as a *flop*, which is short for *floating point operation*. Thus, the computational work required to reduce a system $Ax = b$ of n equations in n unknowns to $Ux = \tilde{b}$ is about $n^3/3$ flops. We show in §2.2 that $Ux = \tilde{b}$ can be solved by *back substitution* using n divisions and about $n^2/2$ flops.

2.2 Back Substitution

Let $U = [u_{ij}]$ be an $n \times n$ nonsingular upper-triangular matrix. That is, $u_{ij} = 0$ for $1 \le j < i \le n$ and $u_{ii} \neq 0$ for $1 \le i \le n$. Then the linear algebraic system $Ux = \tilde{b}$ can be solved easily by *back substitution*, as shown in Table 2.

It is easy to see from Table 2 that back substitution requires n divisions and $n(n-1)/2 \approx n^2/2$ multiplications and subtractions. So the computational work is about $n^2/2$ flops.

2.3 The LU Factorization

Applying Gaussian elimination (GE) as described in §2.1 to solve the linear algebraic system $Ax = b$ of n equations in n unknowns is closely related to computing the LU factorization of the matrix A, where $L_1 = [l_{ij}^{(1)}]$ is a unit-lower-triangular matrix (i.e., $l_{ii}^{(1)} = 1$ for $i = 1, \ldots, n$ and $l_{ij}^{(1)} = 0$ for $1 \le i < j \le n$) and $U_1 = [u_{ij}^{(1)}]$ is an upper-triangular matrix (i.e., $u_{ij}^{(1)} = 0$ for $1 \le j < i \le n$) satisfying

$$A = L_1 U_1. \tag{9}$$

The factorization (9) exists and is unique if and only if all the leading principal minors of A are nonsingular. In this case, it can be shown that the matrix U_1 in (9) is the same as the upper-triangular matrix U produced by GE, and the elements in the strictly lower-triangular part of $L_1 = [l_{ij}^{(1)}]$ satisfy $l_{ij}^{(1)} = m_{ij}$ for $1 \le j < i \le n$, where the m_{ij} are the multipliers used in GE. Moreover, the U_1 in (9) is nonsingular if A is; L_1 is always nonsingular if the factorization exists.

From the discussion in §2.1, it follows that computing the LU factorization of a $n \times n$ matrix A in this way requires $n(n-1)/2 \approx n^2/2$ divisions and $n(2n-1)(n-1)/6 \approx n^3/3$ multiplications and subtractions. Thus, the computational work required to calculate it is about $n^3/3$ flops.

If we need to solve $m > 1$ systems $Ax_i = b_i$, $i = 1, \ldots, m$, (or $AX = B$, where X and $B \in \mathbb{R}^{n \times m}$ or $\mathbb{C}^{n \times m}$), we may obtain significant computational savings by computing the LU factorization of A once only and using the factors L_1 and U_1 to solve each system $Ax_i = b_i$ for $i = 1, \ldots, m$ by first solving $L_1 \tilde{b}_i = b_i$ for $\tilde{b}_i$ by *forward elimination*, as described in §2.4, and then solving $U_1 x_i = \tilde{b}_i$ for x_i by *back substitution*, as outlined in §2.2. This procedure is essentially the same as performing GE to reduce A to U once only, saving the multipliers $\{m_{ik}\}$ used in the process, and then, for each system $Ax_i = b_i$, using the multipliers to perform the same transformation on b_i to produce $\tilde{b}_i$ and solving $Ux_i = \tilde{b}_i$ for x_i by *back substitution*. With either of these procedures, the computational work required to solve all m systems $Ax_i = b_i$ is about $n^3/3 + mn^2$ flops, whereas, if we apply GE as outlined in §2.1 to each system $Ax_i = b_i$, recomputing U each time, the computational work required to solve all m systems $Ax_i = b_i$ is about $m(n^3/3 + n^2)$ flops, which is much greater if m and/or n is large.

Finally, note that we intentionally used the same symbol $\tilde{b}_i$ for the solution of $L\tilde{b}_i = b_i$ and the transformed right side vector produced by GE, since these vectors are identical.

2.4 Forward Elimination

Let $L = [l_{ij}]$ be an $n \times n$ lower-triangular matrix (i.e., $l_{ij} = 0$ for $1 \le i < j \le n$). If L is nonsingular too, then $l_{ii} \neq 0$ for $1 \le i \le n$ and so the linear algebraic system $L\tilde{b} = b$ can be solved easily by *forward elimination*, as shown in Table 3.

It is easy to see from Table 3 that forward elimination requires n divisions and $n(n-1)/2 \approx n^2/2$ multiplications and subtractions. So the computational work is about $n^2/2$ flops.

If $L = [l_{ij}]$ is *unit-lower-triangular* (i.e., $l_{ii} = 1$ for $i = 1, \ldots, n$ as well as L being lower-triangular), as is the case for the L produced by the LU factorization described in §2.3, then the division by l_{ii} in Table 3 is not required, reducing the operation count slightly to about $n^2/2$ flops. However, we have presented the forward elimination procedure in Table 3 with general $l_{ii} \neq 0$, since other schemes, such as the Cholesky factorization described in §2.6, produce a lower-triangular matrix that is typically not unit-lower-triangular.

The name *forward elimination* for this procedure comes from the observation that it is mathematically equivalent to the forward elimination procedure used in GE to eliminate the variables $x_1, \ldots, x_{i-1}$ from equation i of the original system $Ax = b$ to produce the reduced system $Ux = \tilde{b}$.

2.5 Scaling and Pivoting

As noted in §2.1, the simple form of Gaussian elimination (GE) presented there may "break down" at stage k if the pivot $a_{kk}^{(k-1)} = 0$. Moreover, even if $a_{kk}^{(k-1)} \neq 0$, but $|a_{kk}^{(k-1)}| \ll |a_{ik}^{(k-1)}|$ for some $i \in \{k+1, \ldots, n\}$, then $|m_{ik}| = |a_{ik}^{(k-1)}/a_{kk}^{(k-1)}| \gg 1$. So multiplying row k of A_{k-1} by m_{ik} and subtracting it from row i may produce large elements in the resulting row i of A_k, which in turn may produce still larger elements during later stages of the GE process. Since, as noted in §2.8, the bound on the rounding errors in the GE process is proportional to the largest element that occurs in A_k for $k = 0, \ldots, n-1$, creating large elements during the GE reduction process may introduce excessive rounding error into the computation, resulting in an unstable numerical process and destroying the accuracy of the LU factorization and the computed solution x of the linear system $Ax = b$. We present in this section *scaling* and *pivoting* strategies that enhance GE to make it an efficient, robust, reliable numerical method for the solution of linear systems.

Scaling, often called *balancing* or *equilibration*, is the process by which the equations and unknowns of the system $Ax = b$ are scaled in an attempt to reduce the rounding errors incurred in solving the problem and improve its conditioning, as described in §2.8. The effects can be quite dramatic.

Typically, scaling is done by choosing two $n \times n$ diagonal matrices $D_1 = [d_{ij}^{(1)}]$ and $D_2 = [d_{ij}^{(2)}]$ (i.e., $d_{ij}^{(1)} = d_{ij}^{(2)} = 0$ for $i \neq j$) and forming the new system $\hat{A}\hat{x} = \hat{b}$, where $\hat{A} = D_1 A D_2^{-1}$, $\hat{x} = D_2 x$ and $\hat{b} = D_1 b$. Thus, D_1 scales the rows and D_2 scales the unknowns of $Ax = b$, or, equivalently, D_1 scales the rows and D_2^{-1} scales the columns of A. Of course, the solution of $Ax = b$ can be recovered easily from the solution of $\hat{A}\hat{x} = \hat{b}$, since $x = D_2^{-1}\hat{x}$. Moreover, if the diagonal entries $d_{11}^{(1)}, \ldots, d_{nn}^{(1)}$ of D_1 and $d_{11}^{(2)}, \ldots, d_{nn}^{(2)}$ of D_2 are chosen to be powers of the base of the floating-point number system (i.e., powers of 2 for IEEE floating-point arithmetic), then scaling introduces no rounding errors into the computation.

One common technique is to scale the rows only by taking $D_2 = I$ and choosing D_1 so that largest

element in each row of the scaled matrix $\hat{A} = D_1 A$ is about the same size. A slightly more complicated procedure is to scale the rows and columns of A so that the largest element in each row and column of $\hat{A} = D_1 A D_2^{-1}$ is about the same size.

These strategies, although usually helpful, are not fool proof: it is easy to find examples for which row scaling or row and column scaling as described above makes the numerical solution worse. The best strategy is to scale on a problem-by-problem basis depending on what the source problem says about the significance of each coefficient a_{ij} in $A = [a_{ij}]$. See an advanced text such as (Golub and Van Loan 1989) for a more detailed discussion of scaling.

For the remainder of this section, we assume that scaling, if done at all, has already been performed.

The most commonly used pivoting strategy is *partial pivoting*. The only modification required to stage k of GE described in §2.1 to implement GE with partial pivoting is to first search column k of A_{k-1} for the largest element $a_{ik}^{(k-1)}$ on or below the main diagonal. That is, find $i \in \{k, \ldots, n\}$ such that $|a_{ik}^{(k-1)}| \geq |a_{\mu k}^{(k-1)}|$ for $\mu = k, \ldots, n$. Then interchange equations i and k in the reduced system $A_{k-1}x = b_{k-1}$ and proceed with stage k of GE as described in §2.1.

After the equation interchange described above, the *pivot element* $a_{kk}^{(k-1)}$ satisfies $|a_{kk}^{(k-1)}| \geq |a_{ik}^{(k-1)}|$ for $i = k, \ldots, n$. So, if $a_{kk}^{(k-1)} \neq 0$, then the multiplier $m_{ik} = a_{ik}^{(k-1)}/a_{kk}^{(k-1)}$ must satisfy $|m_{ik}| \leq 1$ for $i = k+1, \ldots, n$. Thus no large multipliers can occur in GE with partial pivoting.

On the other hand, if the pivot element $a_{kk}^{(k-1)} = 0$, then $a_{ik}^{(k-1)} = 0$ for $i = k, \ldots, n$, whence A_{k-1} is singular and so A must be too. Thus, GE with partial pivoting never "breaks down" (in exact arithmetic) if A is nonsingular.

Partial pivoting adds a little overhead only to the GE process. At stage k, we must perform $n - k$ comparisons to determine the row i with $|a_{ik}^{(k-1)}| \geq |a_{jk}^{(k-1)}|$ for $j = k, \ldots, n$. Thus, GE with partial pivoting requires a total of $n(n-1)/2 \approx n^2/2$ comparisons. In addition, we must interchange rows i and k if $i > k$, or use some form of indirect addressing if the interchange is not performed explicitly. On the other hand, exactly the same number of arithmetic operations must be executed whether or not pivoting is performed. Therefore, if n is large, the added cost of pivoting is small compared to performing approximately $n^3/3$ flops to reduce A to upper triangular form.

Complete pivoting is similar to partial pivoting except that the search for the pivot at stage k of GE is not restricted to column k of A_{k-1}. Instead, in GE with complete pivoting, we search the $(n-k) \times (n-k)$ lower right block of A_{k-1} for the largest element. That is, find i and $j \in \{k, \ldots, n\}$ such that $|a_{ij}^{(k-1)}| \geq |a_{\mu\nu}^{(k-1)}|$ for $\mu = k, \ldots, n$ and $\nu = k, \ldots, n$. Then interchange equations i and k and variables j and k in the reduced system $A_{k-1}x_{k-1} = b_{k-1}$ and proceed with stage k of GE as described in §2.1. Note that the vector x_{k-1} in the reduced system above is a re-ordered version of the vector of unknowns x in the original system $Ax = b$, incorporating the variable interchanges that have occurred in stages $1, \ldots, k-1$ of GE with complete pivoting.

After the equation and variable interchanges described above, the *pivot element* $a_{kk}^{(k-1)}$ satisfies $|a_{kk}^{(k-1)}| \geq |a_{ik}^{(k-1)}|$ for $i = k, \ldots, n$. So, if $a_{kk}^{(k-1)} \neq 0$, the multiplier $m_{ik} = a_{ik}^{(k-1)}/a_{kk}^{(k-1)}$ must satisfy $|m_{ik}| \leq 1$ for $i = k+1, \ldots, n$, as is the case with partial pivoting. However, with complete pivoting, the multipliers tend to be even smaller than they are with partial pivoting, since the pivots tend to be larger, and so the numerical

solution might suffer less loss of accuracy due to rounding errors, as discussed further in §2.8

After the row and column interchanges in stage k, the pivot element $a_{kk}^{(k-1)} = 0$ only if $a_{ij}^{(k-1)} = 0$ for $i = k, \ldots, n$ and $j = k, \ldots, n$ in which case A_{k-1} is singular and so A must be too. Thus, like GE with partial pivoting, GE with complete pivoting never "breaks down" (in exact arithmetic) if A is nonsingular.

Moreover, if A is singular and $a_{ij}^{(k-1)} = 0$ for $i = k, \ldots, n$ and $j = k, \ldots, n$, then the GE process can be terminated at this stage and the factorization computed so far used to advantage in determining a solution (or approximate solution) to the singular system $Ax = b$. However, this is not as robust a technique as the methods discussed in §4 for over-determined problems, so we do not discuss this further here. The reader interested in this application of GE should consult an advanced text such as (Golub and Van Loan 1989).

Complete pivoting, unlike partial pivoting, adds significantly to the cost of the GE process. At stage k, we must perform $(n-k+1)^2 - 1$ comparisons to determine the row i and column j with $|a_{ij}^{(k-1)}| \geq |a_{\mu\nu}^{(k-1)}|$ for $\mu = k, \ldots, n$ and $\nu = k, \ldots, n$. Thus, GE with partial pivoting requires a total of $n(n-1)(2n+5)/6 \approx n^3/3$ comparisons. On the other hand, exactly the same number of arithmetic operations must be executed whether or not pivoting is performed. So the cost of determining the pivots is comparable to the cost of performing approximately $n^3/3$ flops required to reduce A to upper-triangular form. In addition, we must interchange rows i and k if $i > k$ and columns j and k if $j > k$ or use some form of indirect addressing if the interchange is not performed explicitly. Thus even though GE with complete pivoting has better roundoff error properties than GE with partial pivoting, GE with partial pivoting is used more often in practice.

As is the case for the simple version of GE presented in §2.1, GE with partial or complete pivoting is closely related to computing the LU factorization of the matrix A. However, in this case, we must account for the row or row and column interchanges by extending (9) to

$$P_2 A = L_2 U_2 \tag{10}$$

for partial pivoting and

$$P_3 A Q_3^T = L_3 U_3 \tag{11}$$

for complete pivoting, where P_2 and P_3 are permutation matrices that record the row interchanges performed in GE with partial and complete pivoting, respectively, Q_3^T is a permutation matrix that records the column interchanges performed in GE with complete pivoting, L_2 and L_3 are unit-lower-triangular matrices with the $n-k$ multipliers from stage k of GE with partial and complete pivoting, respectively, in column k below the main diagonal, but permuted according to the row interchanges that occur in stages $k+1, \ldots, n-1$ of GE with partial and complete pivoting, respectively, U_2 and U_3 are the upper-triangular matrices produced by GE with partial and complete pivoting, respectively.

A permutation matrix P has exactly one 1 in each row and column and all other elements equal to 0. It is easy to check that $PP^T = I$ so P is nonsingular and $P^T = P^{-1}$. That is, P is an orthogonal matrix. Also note that we do not need a full $n \times n$ array to store P: the information required to form or multiply by $P = [P_{ij}]$ can be stored in an n-vector $p = [p_i]$, where $p_i = j$ if and only if $P_{ij} = 1$ and $P_{ik} = 0$ for $k \neq j$.

The factorizations (9), (10) and (11) are all called *LU factorizations*; (10) is also called a *PLU factor-*

ization. Unlike (9), the LU factorizations (10) and (11) always exist. P_2, P_3, Q_3 L_2 and L_3 are always nonsingular; U_2 and U_3 are nonsingular if and only if A is nonsingular. Moreover, the factorizations (10) and (11) are unique if there is a well-defined choice for the pivot if more than one element of maximal size occurs in the search for the pivot and if there is a well-defined choice for the multipliers in L if the pivot is zero.

The LU factorization (10) can be used to solve the linear system $Ax = b$ by first computing $\hat{b}_2 = P_2 b$, then solving $L_2 \tilde{b}_2 = \hat{b}_2$ by forward elimination and finally solving $U_2 x = \tilde{b}_2$ by back substitution. The steps are similar if we use the LU factorization (11) instead of (10), except that the back substitution $U_3 \hat{x}_3 = \tilde{b}_3$ yields the permuted vector of unknowns $\hat{x}_3$. The original vector of unknowns x can be recovered by $x = Q_3^T \hat{x}_3$. We have used $\tilde{b}_2$ and $\tilde{b}_3$ for the intermediate results above to emphasize that this is the same as the vector $\tilde{b}$ that is obtained if we perform GE with partial and complete pivoting, respectively, on the original system $Ax = b$.

As noted earlier for (9), if we need to solve $m > 1$ systems $Ax_i = b_i$, $i = 1, \ldots, m$, (or $AX = B$, where X and $B \in \mathbb{R}^{n \times m}$ or $\mathbb{C}^{n \times m}$), we may obtain significant computational savings by computing the LU factorization of A once only. The same observation applies to the LU factorizations (10) and (11).

We end by noting that not having to pivot to ensure numerical stability can be a great advantage in some cases — for example, when factoring a banded or sparse matrix, as described in §2.7. Moreover, there are classes of matrices for which pivoting is not required to ensure numerical stability. Three such classes are complex Hermitian positive-definite matrices, real symmetric positive-definite matrices and diagonally-dominant matrices.

2.6 The Cholesky Factorization

In this subsection, we present the *Cholesky factorization* of a real symmetric positive-definite $n \times n$ matrix A. It is straightforward to modify the scheme for complex Hermitian positive-definite matrices.

Recall that $A \in \mathbb{R}^{n \times n}$ is symmetric if $A = A^T$, where A^T is the transpose of A, and it is positive-definite if $x^T Ax > 0$ for all $x \in \mathbb{R}^n$, $x \neq 0$. The Cholesky factorization exploits these properties of A to compute a lower-triangular matrix L satisfying

$$A = LL^T \tag{12}$$

The similar *LDL factorization* computes a unit-lower-triangular matrix $\tilde{L}$ and a diagonal matrix D satisfying

$$A = \tilde{L} D \tilde{L}^T \tag{13}$$

We present the *dot product* form of the Cholesky factorization in Table 4. It is derived by equating the terms of $A = [a_{ij}]$ to those of LL^T in the order (1,1), (2,1),...,(n,1), (2,2), (3,2),...,(n,2),...,(n,n) and using the lower-triangular structure of $L = [l_{ij}]$ (i.e., $l_{ij} = 0$ for $1 \leq i < j \leq n$). Other forms of the Cholesky factorization are discussed in advanced texts such as (Golub and Van Loan 1989).

It can be shown that, if A is symmetric positive-definite, then

$$a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 > 0$$

(in exact arithmetic) each time this expression is computed in the Cholesky factorization. Therefore, we may take the associated square root to be positive, whence $l_{jj} > 0$ for $j = 1, \ldots, n$. With this convention, the Cholesky factorization is unique.

Moreover, it follows from

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}$$

that

$$a_{jj} = \sum_{k=1}^{j} l_{jk}^2.$$

So the elements in row j of the Cholesky factor L are bounded by $\sqrt{a_{jj}}$ even if we don't pivot. Consequently, as noted in §2.5, it is customary to compute the Cholesky factorization without pivoting.

Note that the method in Table 4 accesses the lower-triangular part of A only, so only those elements need to be stored. Moreover, if we replace l_{jj} and l_{ij} by a_{jj} and a_{ij}, respectively, in Table 4, then the modified algorithm overwrites the lower-triangular part of A with the Cholesky factor L.

A straightforward count of the operations in Table 4 shows that the Cholesky factorization requires n square roots, $n(n-1)/2 \approx n^2/2$ divisions and $n(n-1)(n+1)/6 \approx n^3/6$ multiplications and subtractions. This is approximately half the arithmetic operations required to compute the LU factorization of A. Of course, the storage required for the Cholesky factorization is also about half that required for the LU factorization.

The Cholesky, LDL and LU factorizations are closely related. Let $D_1 = [d_{ij}^{(1)}]$ be the diagonal matrix with the same diagonal elements as $L = [l_{ij}]$ (i.e., $d_{jj}^{(1)} = l_{jj}$ for $j = 1, \ldots, n$ and $d_{ij}^{(1)} = 0$ for $i \neq j$). D_1 is nonsingular, since, as noted above, $l_{jj} > 0$ for $j = 1, \ldots, n$. Moreover, $\tilde{L} = LD_1^{-1}$ is unit lower triangular. If we also let $D = D_1 D_1 = D_1 D_1^T$, then

$$A = LL^T = (\tilde{L}D_1)(\tilde{L}D_1)^T = \tilde{L}D_1 D_1^T \tilde{L}^T = \tilde{L}D\tilde{L}^T,$$

where $\tilde{L}D\tilde{L}^T$ is the LDL factorization of A. Furthermore, if we let $U = D\tilde{L}^T$, then $\tilde{L}U$ is the LU factorization of A.

The LDL factorization can be computed directly by equating the terms of A to those of $\tilde{L}D\tilde{L}^T$, just as we did above for the Cholesky factorization. This leads to a scheme similar to that shown in Table 4, but without any square roots, although it has $n(n-1)/2 \approx n^2/2$ more multiplications. Thus the cost of computing the factorization remains about $n^3/6$ flops and the storage requirement remains about $n^2/2$.

An advantage of the LDL factorization is that it can be applied to a symmetric indefinite matrix. The Cholesky factorization is not applicable in this case, since LL^T is always symmetric positive-semidefinite. However, since LDL^T is always symmetric, pivoting must be restricted to ensure that the re-ordered matrix

PAQ^T is symmetric. The simplest way to maintain symmetry is to use *symmetric pivoting* in which $Q = P$. This, though, restricts the choice of the pivot at stage k of the LDL factorization to a_{jj} for $j = k, \ldots, n$. As a result, in some cases, the LDL factorization may incur much more rounding error than GE with partial or complete pivoting.

2.7 Banded and Sparse Matrices

Significant savings in both computational work and storage can often be obtained in solving $Ax = b$ by taking advantage of zeros in the coefficient matrix A. We outline in this subsection how these savings may be realized for banded and more general sparse matrices.

To begin, note that an $n \times n$ matrix A is said to be *sparse* if the number of nonzero elements in A is much less than n^2, the total number of elements in A. *Banded matrices* are an important subclass of sparse matrices in which the nonzero elements of the matrix are restricted to a band around the main diagonal of the matrix. The *lower bandwidth* of a matrix $A = [a_{ij}]$ is the smallest integer p such that $a_{ij} = 0$ for $i - j > p$, the *upper bandwidth* of A is the smallest integer q such that $a_{ij} = 0$ for $j - i > q$, and the *bandwidth* of A is $1 + p + q$. Clearly, if p and $q \ll n$, then A is sparse, since A has at most $(1 + p + q)n \ll n^2$ nonzero elements.

Banded and more general sparse matrices arise in many important applications. For example, quadratic spline interpolation, as described in §8.6.2, requires the solution of a linear systems $Tc = g$, where T is a tridiagonal matrix (i.e., banded with $p = q = 1$) and c is the vector of coefficients for the quadratic spline interpolant. Banded matrices also arise in the solution of boundary value problems for ordinary differential equations. See §11.3.1 for an example of a system $Tu = g$, where T is a symmetric positive-definite tridiagonal matrix. More general sparse matrices arise in the numerical solution of partial differential equations. See §11.3.2 for an example of the matrix associated with the standard 5-point difference scheme for Poisson's equation.

If A is large and sparse, it is common to store only the nonzero elements of A, since this greatly reduces the storage requirements. This is easy to do if A is banded, since we can map the elements from the band of A to a $(1 + p + q) \times n$ array representing A in a *packed format*, several of which are commonly used in practice. If $A = [a_{ij}]$ is a general sparse matrix, then we require a more general *sparse matrix data structure* which stores each nonzero element a_{ij} of A along with some information used to recover the indices i and j.

We consider Gaussian elimination (GE) for a banded matrix first. To begin, note that the reduction process described in §2.1 maintains the band structure of A. In particular, $a_{ik}^{(k-1)} = 0$ for $i - k > p$, whence the multipliers $m_{ik} = a_{ik}^{(k-1)}/a_{kk}^{(k-1)}$ in Table 1 need to be calculated for $i = k + 1, \ldots \min(k + p, n)$ only and the i loop can be changed accordingly. Similarly $a_{kj}^{(k-1)} = 0$ for $j - k > q$, so the reduction $a_{ij}^{(k)} = a_{ij}^{(k-1)} - m_{ij} \cdot a_{kj}^{(k-1)}$ in Table 1 needs to be calculated for $j = k + 1, \ldots, \min(k + q, n)$ only and the j loop can be changed accordingly. It therefore follows from a straightforward operation count that GE modified as described above for banded matrices requires $np - p(p + 1)/2 \approx np$ divisions to compute the multipliers m_{ik}, either $npq - p(3q^2 + 3q + p^2 - 1)/6 \approx npq$ if $p \leq q$ or $npq - q(3p^2 + 3p + q^2 - 1)/6 \approx npq$ if $p \geq q$ multiplications and subtractions to compute the coefficients of U, and $np - p(p + 1)/2 \approx np$ multiplications

and subtractions to compute the coefficients of $\tilde{b}$. Furthermore, note that, if we use this modified GE procedure to compute the LU factorization of A, then the lower-triangular matrix L has lower bandwidth p and the upper-triangular matrix U has upper bandwidth q. As noted in §2.1, it is common to overwrite A with L and U. This can be done even if A is stored in packed format, thereby achieving significant reduction in storage requirements.

The back substitution method shown in Table 2 and the forward elimination method shown in Table 3 can be modified similarly so that each requires n divisions, the back substitution method requires $nq - q(q+1)/2 \approx nq$ multiplications and subtractions while the forward elimination method requires $np - p(p+1)/2 \approx np$ multiplications and subtractions. In addition, recall that the n divisions are not needed in forward elimination if L is unit-lower-triangular, as is the case for the modified LU factorization described here.

A similar modification of the Cholesky method shown in Table 4 results in a procedure that requires n square roots, $np - p(p+1)/2 \approx np$ divisions and $(n-p)p(p+1)/2 + (p-1)p(p+1)/6 \approx np^2/2$ multiplications and subtractions. In deriving these operation counts, we used $p = q$, since the matrix A must be symmetric for the Cholesky factorization to be applicable. Moreover, the Cholesky factor L has lower bandwidth p. Thus, as for the general case, the Cholesky factorization of a band matrix requires about half as many arithmetic operations and about half as much storage as the LU factorization, since packed storage can also be used for the Cholesky factor L.

If partial pivoting is used in the LU factorization of A, then the upper bandwidth of U may increase to $p + q$. The associated matrix L is a permuted version of a lower triangular matrix with lower bandwidth p. Both factors L and U can be stored in packed format in a $(1 + 2p + q) \times n$ array. The operation count for the LU factorization, forward elimination and back solve is the same as though A were a banded matrix with upper bandwidth $p + q$ and lower bandwidth p. Thus, if $p > q$, both computational work and storage can be saved by factoring A^T instead of A and using the LU factors of A^T to solve $Ax = b$.

If complete pivoting is used in the LU factorization of A, then L and U may fill-in so much that there is little advantage to using a band solver.

Consequently, it is advantageous not to pivot when factoring a band matrix, provided this does not lead to an unacceptable growth in rounding errors. As noted in §2.5, it is not necessary to pivot for numerical stability if A is complex Hermitian positive-definite, real symmetric positive-definite or column-diagonally-dominant. If pivoting is required, it is advantageous to use partial, rather than complete, pivoting, again provided this does not lead to an unacceptable growth in rounding errors.

Extending GE to take advantage of the zeros in a general sparse matrix is considerably more complicated than for banded matrices. The difficulty is that, when row k of A_{k-1} is multiplied by m_{ik} and added to row i of A_{k-1} to eliminate $a_{ik}^{(k-1)}$ in stage k of GE, as described in §2.1, some zero elements in row i of A_{k-1} may become nonzero in the resulting row i of A_k. These elements are said to *fill in* and are collectively referred to as *fill-in* or *fill*.

However, pivoting can often greatly reduce the amount of fill-in. To see how this comes about, the

interested reader may wish to work through an example with the *arrow-head* matrix

$$A = \begin{pmatrix} 5 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Since A is a real symmetric positive-definite matrix, there is no need to pivot for numerical stability. It is easy to see that the Cholesky and LU factors of A completely fill in. Interchanging the first and last rows and the first and last columns of A corresponds to forming the permuted matrix $B = PAP^T$, where $P = I - [1,0,0,0,-1]^T[1,0,0,0,-1]$ is a permutation matrix. Since B is also a real symmetric positive-definite matrix, there is no need to pivot for numerical stability when factoring B. However, in this case, the Cholesky and LU factors of B suffer no fill in at all. This small example can be generalized easily to arbitrarily large arrow-head matrices having the similar properties that the LU factors of A completely fill-in while those of the permuted matrix $B = PAP^T$ suffer no fill-in at all.

If we use an appropriate sparse-matrix data structure to store only the nonzeros elements of a sparse matrix A and its LU factors, then reducing the fill-in reduces both the storage needed for the LU factors and the computational work required to calculate them. It also reduces the computational work required for forward elimination and back substitution, since the methods shown in Table 2 and 3 can be modified easily so that they use the nonzero elements in L and U only, thereby avoiding multiplications by zero and the associated subtractions.

Therefore, the goal in a sparse LU or Cholesky factorization is to re-order the rows and columns of A to reduce the amount of fill-in. If we need to pivot to ensure numerical stability, then this might conflict with pivoting to reduce fill-in. Unfortunately, even without this complication, finding the optimal re-ordering to minimize the fill-in is computationally too expensive to be feasible in general. There are, though, many good *heuristic* methods to re-order the rows and columns of A that greatly reduces the fill-in and computational work in many important cases. The reader seeking a more complete description of sparse matrix factorizations should consult a text on this topic, such as (Duff, Erisman and Reid 1986) or (George and Liu 1981).

We end this subsection with an example illustrating the importance of exploiting the zeros in a sparse matrix A. Consider the linear system derived in §11.3.2 by discretizing Poisson's equation on an $m \times m$ grid. A is an $n \times n$ symmetric positive-definite matrix with $n = m^2$ and $5m^2 - 4m \approx 5m^2 = 5n$ nonzero elements, out of a total of n^2 elements in A. So, if m is large, A is very sparse. Moreover, if we use the *natural ordering* for the equations and variables in the system, then A, as shown in §11.3.2, is a banded matrix with lower and upper bandwidth $m = \sqrt{n}$.

The computational work and storage required to solve this linear system are shown in Table 5. We consider three cases: (1) dense, the zeros in A are not exploited at all; (2) banded, we use a band solver that exploits the band structure of A; (3) sparse, we use a general sparse solver together with the the *nested dissection ordering* (see (George and Liu 1981)) for the equations and variables in the system. The columns

labeled factor and solve, respectively, give the approximate number of flops needed to compute the Cholesky factorization of the matrix A and to solve the linear system given the factors. The columns labeled store A and store L, respectively, give the approximate number of storage locations needed to store A and its Cholesky factor L.

2.8 Rounding Errors, Condition Numbers and Error Bounds

In this subsection, we consider the effects of rounding errors in solving $Ax = b$. In doing so, we use vector and matrix norms extensively. Therefore, we recommend that, if you are not acquainted with norms, you familiarize yourself with this topic before reading this subsection. Most introductory numerical methods texts or advanced books on numerical linear algebra contain a section on vector and matrix norms.

The analysis of the effects of rounding errors in solving $Ax = b$ usually proceeds in two stages. First we establish that the computed solution $\tilde{x}$ is the exact solution of a perturbed system

$$(A + E)\tilde{x} = b + r \tag{14}$$

with bounds on the size of E and r. Then we use (14) together with bounds on the size of A and A^{-1} to bound the error $x - \tilde{x}$.

The first step is called a *backward error analysis*, since it casts the error in the solution back onto the problem and allows us to relate the effects of rounding errors in the computed solution $\tilde{x}$ to other errors in the problem, such as measurement errors in determining the coefficients of A and b. Another advantage of proceeding in this two stage fashion is that, if $\tilde{x}$ is not sufficiently accurate, it allows us to determine whether this is because the numerical method is faulty or whether the problem itself is unstable.

Throughout this section we assume that A is an $n \times n$ nonsingular matrix and $nu < 0.1$, where u is the *relative roundoff error bound* for the machine arithmetic used in the computation (see §1.2).

If we use Gaussian elimination (GE) with either partial or complete pivoting together with forward elimination and back substitution to solve $Ax = b$, then it can be shown that the computed solution $\tilde{x}$ satisfies (14) with $r = 0$ and

$$\|E\|_\infty \leq 8n^3\gamma\|A\|_\infty u + O(u^2), \tag{15}$$

where

$$\gamma = \max_{i,j,k} \frac{|a_{ij}^{(k)}|}{\|A\|_\infty}$$

is the *growth factor* and $A_k = [a_{ij}^{(k)}]$ for $k = 0, \ldots, n-1$ are the intermediate reduced matrices produced during the GE process (see §2.1 and §2.5). It can be shown that $\gamma \leq 2^{n-1}$ for GE with partial pivoting and $\gamma \leq \sqrt{n(2 \cdot 3^{1/2} \cdot 4^{1/3} \cdots n^{1/(n-1)})} \ll 2^{n-1}$ for GE with complete pivoting. Moreover, the former upper bound can be achieved. However, for GE with both partial and complete pivoting, the actual error incurred is usually much smaller than the bound (15) suggests: it is typically the case that $\|E\|_\infty \propto \|A\|_\infty u$. Thus, GE with partial pivoting usually produces a computed solution with a small backward error, but, unlike GE with complete pivoting, there is no guarantee that this will be the case.

If A is column diagonally dominant, then applying GE without pivoting to solve $Ax = b$ is effectively the same as applying GE with partial pivoting to solve this system. Therefore, all the remarks above for GE with partial pivoting apply in this special case.

If we use the Cholesky factorization without pivoting together with forward elimination and back substitution to solve $Ax = b$, where A is a symmetric positive-definite matrix, then it can be shown that the computed solution $\tilde{x}$ satisfies (14) with $r = 0$ and

$$\|E\|_2 \leq c_n \|A\|_2 u \tag{16}$$

where c_n is a constant of moderate size that depends on n only. Thus the Cholesky factorization without pivoting produces a solution with a small backward error in all cases.

Similar bounds on the backward error for other factorizations and matrices with special properties, such as symmetric or band matrices, can be found in advanced texts, such as (Golub and Van Loan 1989).

It is also worth noting that we can easily compute an a posteriori backward error estimate of the form (14) with $E = 0$ by calculating the *residual* $r = A\tilde{x} - b$ after computing $\tilde{x}$. Typically $\|r\|_\infty \propto \|b\|_\infty u$ if GE with partial or complete pivoting or the Cholesky factorization is used to compute $\tilde{x}$.

Now we use (14) together with bounds on the size of A and A^{-1} to bound the error $x - \tilde{x}$. To do so, we first introduce the *condition number* $\kappa(A) = \|A\|\|A^{-1}\|$ associated with the problem $Ax = b$. Although $\kappa(A)$ clearly depends on the matrix norm used, it is roughly of the same magnitude for all the commonly used norms, and it is the magnitude only of $\kappa(A)$ that is important here. Moreover, for the result below to hold, we require only that the matrix norm associated with $\kappa(A)$ is sub-multiplicative (i.e., $\|AB\| \leq \|A\|\|B\|$ for all A and $B \in \mathbb{C}^{n\times n}$) and that it is consistent with the vector norm used (i.e., $\|Av\| \leq \|A\|\|v\|$ for all $A \in \mathbb{C}^{n\times n}$ and $v \in \mathbb{C}^n$). It follows immediately from these two properties that $\kappa(A) \geq 1$ for all $A \in \mathbb{C}^{n\times n}$. More importantly, it can be shown that, if $\|E\|/\|A\| \leq \delta$, $\|r\|/\|b\| \leq \delta$ and $\delta\kappa(A) = r < 1$, then $A + E$ is nonsingular and

$$\frac{\|x - \tilde{x}\|}{\|x\|} \leq \frac{2\delta}{1 - r}\kappa(A). \tag{17}$$

Moreover, for any given A, there are some b, E and r for which $\|x - \tilde{x}\|/\|x\|$ is as large as the right side of (17) suggests it might be, although this is not the case for all b, E and r. Thus we see that, if $\kappa(A)$ is not too large, then small relative errors $\|E\|/\|A\|$ and $\|r\|/\|b\|$ ensure a small relative $\|x - \tilde{x}\|/\|x\|$, so the problem is *well-conditioned*. On the other hand, if $\kappa(A)$ is large, then $\|x - \tilde{x}\|/\|x\|$ might be large even though $\|E\|/\|A\|$ and $\|r\|/\|b\|$ are small, so the problem is *ill-conditioned*. Thus, as the name suggests, the condition number $\kappa(A)$ gives a good measure of the conditioning — or stability — of the problem $Ax = b$.

Combining the discussion above with the earlier observation that typically $\|r\|_\infty \propto \|b\|_\infty u$ if GE with partial or complete pivoting or the Cholesky factorization is used to compute $\tilde{x}$, we get the general rule of thumb that, if $u \approx 10^{-d}$ and $\kappa_\infty = \|A\|_\infty\|A^{-1}\|_\infty \approx 10^q$, then $\tilde{x}$ contains about $d - q$ correct digits.

Many routines for solving linear systems provide an estimate of $\kappa(A)$, although most do not compute $\|A\|\|A^{-1}\|$ directly, since they do not compute A^{-1}.

There are many other useful inequalities of the form (17). The interested reader should consult an

advanced text, such as (Golub and Van Loan 1989).

2.9 Iterative Improvement

The basis of *iterative improvement* is the observation that, if x_1 is an approximate solution to $Ax = b$, we can form the residual $r_1 = b - Ax_1$, which satisfies $r_1 = A(x - x_1)$, and then solve $Ad_1 = r_1$ for the difference $d_1 = x - x_1$ and finally compute the improved solution $x_2 = x_1 + d_1$. In exact arithmetic, $x_2 = x$, but, in floating-point arithmetic, $x_2 \neq x$ normally. So we can repeat the process using x_2 in place of x_1 to form another improved solution x_3, and so on. Moreover, if we have factored A to compute x_1, then, as noted in §2.3 and §2.5, there is relatively little extra computational work required to compute a few iterations of iterative improvement.

The catch here is that, as noted in §2.8, typically $\|r\|_\infty \propto \|b\|_\infty u$ if GE with partial or complete pivoting or the Cholesky factorization is used to compute $\tilde{x}$. So, if we compute $r_1 = b - Ax_1$ in the same precision, then r_1 will contain few if any correct digits. Consequently, using it in iterative improvement usually does not lead to a reduction of the error in x_2, although it may lead to a smaller E in (14) in some cases. However, if we compute $r_k = b - Ax_k$ in double precision for $k = 1, 2, \ldots$, then iterative improvement may be quite effective. Roughly speaking, if the relative roundoff error bound $u \approx 10^{-d}$ and the condition number $\kappa(A) = \|A\|\|A^{-1}\| \approx 10^q$, then after k iterations of iterative improvement the computed solution x_k typically has about $\min(d, k(d-q))$ correct digits. Thus, if $\kappa(A)$ is large, but not too large, and, as a result, the initial computed solution x_1 is inaccurate, but not completely wrong, then iterative improvement can be used to obtain almost full single-precision accuracy in the solution.

The discussion above can be made more rigorous by noting that iterative improvement is a basic iterative method of the form (19)–(20) for solving $Ax = b$ and applying the analysis in §3.1.

3 The Iterative Solution of Linear Algebraic Systems

In this section, we consider *iterative* methods for the numerical solution of linear algebraic systems of the form $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ (or $\mathbb{C}^{n \times n}$) is a nonsingular matrix, x and $b \in \mathbb{R}^n$ (or $\mathbb{C}^n$). Such schemes compute a sequence of approximations $x_1, x_2, \ldots$ to x in the hope that $x_k \to x$ as $k \to \infty$. Direct methods for solving $Ax = b$ are considered in §2.

Iterative methods are most frequently used when A is large and sparse, but not banded with a small bandwidth. Such matrices arise frequently in the numerical solution of partial differential equations (PDEs). See, for example, the matrix shown in §11.3.2 that is associated with the standard 5-point difference scheme for Poisson's equation. In many such cases, iterative methods are more efficient than direct methods for solving $Ax = b$: they usually use far less storage and often require significantly less computational work as well.

We discuss *basic iterative methods* in §3.1 and the *conjugate gradient acceleration* of these schemes in §3.2. A more complete description and analysis of these methods and other iterative schemes is provided in (Axelsson 1994; Golub and Van Loan 1989; Hageman and Young 1981; Young 1971; Varga 1962). See §13 for

a discussion of sources of high-quality numerical software for solving systems of linear algebraic equations.

We discuss multigrid methods in §11.8, because these iterative schemes are so closely tied to the PDE that gives rise to the linear system $Ax = b$ to which they are applied.

3.1 Basic Iterative Methods

Many iterative methods for solving $Ax = b$ are based on splitting the matrix A into two parts, M and N, such that $A = M - N$ with M nonsingular. M is frequently called the *splitting matrix*. Starting from an initial guess x_0 for x, we compute $x_1, x_2, \ldots$ recursively from

$$Mx_{k+1} = Nx_k + b. \tag{18}$$

We call such a scheme a *basic iterative method*, but it is often also referred to as a *linear stationary method of the first degree*.

Since $N = M - A$, (18) can be rewritten as

$$Mx_{k+1} = (M - A)x_k + b = Mx_k + (b - Ax_k)$$

which is equivalent to

$$Md_k = r_k \tag{19}$$

$$x_{k+1} = x_k + d_k \tag{20}$$

where $r_k = b - Ax_k$ is the residual at iteration k. Although (18) and (19)–(20) are mathematically equivalent, it might be computationally more effective to implement a method in one form than the other.

Clearly, for either (18) or (19)–(20) to be effective,

(1) it must be much easier to solve systems with M than with A, and

(2) the iterates $x_1, x_2, \ldots$ generated by (18) or (19)–(20) must converge quickly to x, the solution of $Ax = b$.

To address point (2), first note that, since $Ax = b$ and $A = M - N$, $Mx = Nx + b$. So, if the sequence $x_1, x_2, \ldots$ converges, it must converge to x. To determine whether the sequence $x_1, x_2, \ldots$ converges and, if so, how fast, subtract (18) from $Mx = Nx + b$ and note that the error $e_k = x - x_k$ satisfies the recurrence $Me_{k+1} = Ne_k$, or equivalently $e_{k+1} = Ge_k$, where

$$G = M^{-1}N = I - M^{-1}A$$

is the associated *iteration matrix*. So

$$e_k = G^k e_0. \tag{21}$$

Using (21), we can show that, starting from any initial guess x_0, the sequence $x_1, x_2, \ldots$ generated by (18)

converges to x if and only if $\rho(G) < 1$, where

$$\rho(G) = \max\{|\lambda| : \lambda \text{ an eigenvalue of } G\}$$

is the *spectral radius* of G. Moreover, $\rho(G)$ is the "asymptotically average" amount by which the error e_k decreases at each iteration. Consequently, $(\log \epsilon)/(\log \rho(G))$ is a rough estimate of the number of iterations of (18) required to reduce the initial error e_0 by a factor ϵ. Thus, it is common to define

$$R(G) = -\log \rho(G) \tag{22}$$

to be the *rate of convergence* (sometimes called the *asymptotic rate of convergence* or the *asymptotic average rate of convergence*) of the iteration (18).

One useful general result is that, if $A = M - N$ is Hermitian positive-definite and if the Hermitian matrix $M^H + N$ is positive-definite too, then $\rho(G) < 1$ and the associated iteration (18) converges.

Possibly the simplest iterative scheme is the *RF method* (a variant of Richardson's method) for which $M = I$ and $N = I - A$, whence (18) reduces to

$$x_{k+1} = x_k + r_k$$

where $r_k = b - Ax_k$ is the residual at iteration k. From the general discussion above, it follows that this scheme converges if and only if $\rho(I - A) < 1$. Because of this severe constraint on convergence, this scheme is not often effective in its own right, but it can be used productively as the basis for polynomial acceleration, as discussed in §3.2.

We describe the Jacobi, Gauss-Seidel, SOR and SSOR methods next, and then consider their convergence. In describing them, we use the notation $A = D - L - U$, where D is assumed to be nonsingular and consists of the diagonal elements of A for the point variant of each method or the diagonal submatrices of A for the block variant. L and U are the strictly lower and upper triangular parts of A, respectively, either point or block, as the case may be. Typically, the block variant of each method converges faster than the point version, but requires more computational work per iteration. Thus, it is usually not clear without additional analysis which variant will be more effective.

The *Jacobi iteration* takes $M_J = D$ and $N_J = L + U$, resulting in the recurrence

$$Dx_{k+1} = (L + U)x_k + b. \tag{23}$$

The associated iteration matrix is $G_J = D^{-1}(L + U)$. The *Gauss-Seidel iteration* takes $M_{GS} = D - L$ and $N_{GS} = U$, resulting in the recurrence

$$(D - L)x_{k+1} = Ux_k + b. \tag{24}$$

The associated iteration matrix is $G_{GS} = (D - L)^{-1}U$. Although (24) may at first appear a little more

complicated than (23), it is in fact easier to implement in practice on a sequential machine, since one can overwrite x_k when computing x_{k+1} in (24), whereas this is generally not possible for (23). However, for the Gauss-Seidel iteration, the j^{th} component of x_{k+1} might depend on the i^{th} component of x_{k+1} for $i < j$, because of the factor L on the left side of (24). This often inhibits vectorization and parallelization of the Gauss-Seidel iteration. Note that the Jacobi iteration has no such dependence, and so might be more effective on a vector or parallel machine.

Relaxation methods for $Ax = b$ can be written in the form

$$x_{k+1} = x_k + \omega(\hat{x}_{k+1} - x_k) \tag{25}$$

where $\omega \neq 0$ is the relaxation parameter and $\hat{x}_{k+1}$ is computed from x_k by some other iterative method. The best known of these schemes is *successive over relaxation* (SOR) for which

$$D\hat{x}_{k+1} = Lx_{k+1} + Ux_k + b. \tag{26}$$

Equations (25) and (26) can be combined to give

$$\left(\frac{1}{\omega}D - L\right) x_{k+1} = \left(\frac{1-\omega}{\omega}D + U\right) x_k + b \tag{27}$$

which is an iteration of the form (18) with $M_{SOR}(\omega) = \frac{1}{\omega}D - L$ and $N_{SOR}(\omega) = \frac{1-\omega}{\omega}D + U$. It follows immediately from (24) and (27) that the SOR iteration reduces to the Gauss-Seidel method if $\omega = 1$. Moreover, because of the similarity between (24) and (27), the SOR iteration shares with the Gauss-Seidel method the implementation advantages and disadvantages noted above.

Over relaxation corresponds to choosing $\omega > 1$ in (25) or (27), while under relaxation corresponds to choosing $\omega \in (0, 1)$. Historically, $\omega > 1$ was used in SOR for the solution of elliptic PDEs — hence the name successive *over* relaxation — but under relaxation is more effective for some problems. See for example (Young 1971) for a more complete discussion.

The *symmetric SOR* (SSOR) method takes one half step of SOR with the equations solved in the standard order followed by one half step of SOR with the equations solved in the reverse order:

$$\left(\frac{1}{\omega}D - L\right) x_{k+1/2} = \left(\frac{1-\omega}{\omega}D + U\right) x_k + b \tag{28}$$

$$\left(\frac{1}{\omega}D - U\right) x_{k+1} = \left(\frac{1-\omega}{\omega}D + L\right) x_{k+1/2} + b. \tag{29}$$

These two half steps can be combined into one step of the form (18) with

$$M_{SSOR}(\omega) = \frac{\omega}{2-\omega}\left(\frac{1}{\omega}D - L\right) D^{-1} \left(\frac{1}{\omega}D - U\right)$$

and

$$N_{SSOR}(\omega) = \frac{(\omega-1)^2}{\omega(2-\omega)}D + \frac{1-\omega}{2-\omega}(L+U) + \frac{\omega}{2-\omega}LD^{-1}U.$$

Note that, if $A = D - L - U$ is a real, symmetric, positive-definite matrix and $\omega \in (0,2)$, then $M_{SSOR}(\omega)$ is a real, symmetric, positive-definite matrix too, since, in this case, $\omega/(2-\omega) > 0$, both D and D^{-1} are symmetric positive-definite and $\left(\frac{1}{\omega}D - L\right) = \left(\frac{1}{\omega}D - U\right)^T$ is nonsingular. This property of $M_{SSOR}(\omega)$ plays an important role in the effective acceleration of the SSOR iteration, as discussed in §3.2.

We now consider the convergence of the Jacobi, Gauss-Seidel, SOR and SSOR methods. It is easy to show that the Jacobi iteration (23) converges if A is either row or column diagonally dominant. It can also be shown that the Gauss-Seidel iteration (24) converges if A is Hermitian positive-definite. Furthermore, if A is *consistently ordered*, a property enjoyed by a large class of matrices, including many that arise from the discretization of PDEs (see for example (Young 1971) for details), it can be shown that the Gauss-Seidel iteration converges twice as fast as the Jacobi iteration if either one converges.

The iteration matrix associated with the SOR iteration (27) is

$$G_{SOR}(\omega) = (M_{SOR}(\omega))^{-1} N_{SOR}(\omega) = \left(\frac{1}{\omega}D - L\right)^{-1} \left(\frac{1-\omega}{\omega}D + U\right).$$

For any nonsingular A with nonsingular D, it can be shown that $\rho(G_{SOR}(\omega)) \geq |\omega - 1|$ with equality possible if and only if all eigenvalues of $G_{SOR}(\omega)$ have magnitude $|\omega - 1|$. So, if $\omega \in \mathbb{R}$, as is normally the case, a necessary condition for the convergence of SOR is $\omega \in (0,2)$. It can also be shown that, if A is Hermitian, D is positive-definite and $\omega \in \mathbb{R}$, then the SOR iteration converges if and only if A is positive-definite and $\omega \in (0,2)$. Furthermore, if A is consistently ordered and all the eigenvalues of $G_J = D^{-1}(L+U)$ are real and lie in $(-1,1)$, then the optimal choice of the SOR parameter ω is

$$\omega_0 = \frac{2}{1 + \sqrt{1 - (\rho(G_J))^2}} \in (1,2)$$

and

$$\begin{aligned} \rho(G_{SOR}(\omega_0)) &= \omega_0 - 1 = \left(\frac{\rho(G_J)}{1 + \sqrt{1 - (\rho(G_J))^2}}\right)^2 = \min_{\omega} \rho(G_{SOR}(\omega)) \\ &< \rho(G_1) = \rho(G_{GS}) = (\rho(G_J))^2 < \rho(G_J). \end{aligned} \tag{30}$$

(Recall G_J and G_{GS} are the iteration matrices associated with the Jacobi and Gauss-Seidel iterations, respectively, defined above.) In many cases, though, it is not convenient to calculate the optimal ω. Hageman and Young (1981) discuss heuristics for choosing a "good" ω.

If A is a real, symmetric, positive-definite matrix, then the SSOR iteration (28)–(29) converges for any $\omega \in (0,2)$. Moreover, determining the precise value of the optimal ω is not nearly as critical for SSOR as it is for SOR, since, unlike SOR, the rate of convergence of SSOR is relatively insensitive to the choice of ω. However, for SSOR to be effective,

$$\rho(D^{-1}LD^{-1}U) \leq 1/4,$$

should be satisfied — or nearly so. If this is the case, then a good value for ω is

$$\omega_1 = \frac{2}{1 + \sqrt{2(1 - \rho(G_J))}}$$

and

$$\rho(G_{SSOR}(\omega_1)) \leq \frac{1-\sqrt{\frac{1-\rho(G_J)}{2}}}{1+\sqrt{\frac{1-\rho(G_J)}{2}}} \tag{31}$$

where $G_{SSOR}(\omega) = (M_{SSOR}(\omega))^{-1} N_{SSOR}(\omega)$ is the SSOR iteration matrix.

For a large class of problems, including many that arise from the discretization of elliptic PDEs, $\rho(G_J) = 1 - \epsilon$ for some ϵ satisfying $0 < \epsilon \ll 1$. For such problems, if (30) is valid and (31) holds with $=$ in place of $\leq$, then

$$\begin{aligned}
\rho(G_J) &= 1-\epsilon, \\
\rho(G_{GS}) &= (\rho(G_J))^2 \approx 1-2\epsilon, \\
\rho(G_{SSOR}(\omega_1)) &\approx 1-2\sqrt{\epsilon/2}, \\
\rho(G_{SOR}(\omega_0)) &\approx 1-2\sqrt{2\epsilon},
\end{aligned}$$

whence the rates of convergence for these schemes are

$$\begin{aligned}
R(G_J) &\approx \epsilon, \\
R(G_{GS}) &= 2R(G_J) \approx 2\epsilon, \\
R(G_{SSOR}(\omega_1)) &\approx 2\sqrt{\epsilon/2}, \\
R(G_{SOR}(\omega_0)) &\approx 2\sqrt{2\epsilon} \approx 2R(G_{SSOR}(\omega_1)),
\end{aligned}$$

showing that the Gauss-Seidel iteration converges twice as fast as the Jacobi iteration, the SOR iteration converges about twice as fast as the SSOR iteration and the SOR and SSOR iterations converge much faster than either the Gauss-Seidel or Jacobi iteration. However, the SSOR iteration often has the advantage, not normally shared by the SOR method, that it can be accelerated effectively, as discussed in §3.2.

Another class of basic iterative methods is based on *incomplete factorizations*. For brevity, we describe only the subclass of *incomplete Cholesky factorizations* (ICFs) here; the other schemes are similar. See an advanced text, such as (Axelsson 1994), for details.

For an ICF to be effective, A should be symmetric positive-definite (or nearly so), large and sparse. If A is banded, then the band containing the nonzeros should also be sparse, as is the case for the discretization of Poisson's equation shown in §11.3.2. The general idea behind the ICFs is to compute a lower triangular matrix L_{ICF} such that $M_{ICF} = L_{ICF}L_{ICF}^T$ is in some sense close to A and L_{ICF} is much sparser than L, the true Cholesky factor of A. Then employ the iteration (19)–(20) to compute a sequence of approximations $x_1, x_2, \ldots$ to x, the solution of $Ax = b$. Note that (19) can be solved efficiently by forward elimination and back substitution as described in §2.4 and §2.2, respectively, since the factorization $M_{ICF} = L_{ICF}L_{ICF}^T$ is known. This scheme, or an accelerated variant of it, is often very effective if it converges rapidly and L_{ICF} is much sparser than L.

A simple, but often effective, way of computing $L_{ICF} = [l_{ij}]$ is to apply the Cholesky factorization

described in Table 4, but to set $l_{ij} = 0$ whenever $a_{ij} = 0$, where $A = [a_{ij}]$. Thus, L_{ICF} has the same sparsity pattern as the lower triangular part of A, whereas the true Cholesky factor L of A might suffer significant fillin, as described in §2.7. Unfortunately, this simple ICF is not always stable.

As noted in §2.9, iterative improvement is a basic iterative method of this form, although, for iterative improvement, the error $N = M - A$ is due entirely to rounding errors, whereas, for the incomplete factorizations considered here, the error $N = M - A$ is typically also due to dropping elements from the factors of M to reduce fillin.

For a more complete discussion of ICFs and other incomplete factorizations, their convergence properties, and their potential for acceleration, see an advanced text such as (Axelsson 1994).

We end this section with a brief discussion of *alternating direction implicit* (ADI) methods. A typical example of a scheme of this class is the *Peaceman-Rachford* method

$$(H + \alpha_n I)x_{n+1/2} = b - (V - \alpha_n I)x_n \tag{32}$$

$$(V + \alpha'_n I)x_{n+1} = b - (H - \alpha'_n I)x_{n+1/2} \tag{33}$$

where $A = H + V$, $\alpha_n > 0$, $\alpha'_n > 0$ and A, H and V are real, symmetric, positive-definite matrices. For many problems, it is possible to choose H, V and $\{\alpha_n, \alpha'_n\}$, so that the iteration (32)–(33) converges rapidly and it is much cheaper to solve (32) and (33) than it is to solve $Ax = b$. For example, for the standard 5-point discretization of a separable two-dimensional elliptic PDE, H and V can be chosen to be essentially tridiagonal and the rate of convergence of (32)–(33) is proportional to $1/\log h^{-1}$, where h is the meshsize used in the discretization. In contrast, the rate of convergence for SOR with the optimal ω is proportional to h. Note $h \ll 1/\log h^{-1}$ for $0 < h \ll 1$, supporting the empirical evidence that ADI schemes converge much more rapidly than SOR for many problems. However, ADI schemes are not applicable to as wide a class of problems as SOR is.

For a more complete discussion of ADI schemes, see an advanced text such as (Varga 1962) or (Young 1971).

3.2 The Conjugate Gradient Method

The *conjugate gradient* (CG) *method* for the solution of the linear system $Ax = b$ is a member of a broader class of methods often called polynomial acceleration techniques or Krylov subspace methods. (The basis of these names is explained below.) Although many schemes in this broader class are very useful in practice, we discuss CG only here, but note that several of these schemes, including Chebyshev acceleration and GMRES, apply to more general problems than CG. The interested reader should consult an advanced text such as (Axelsson 1994; Golub and Van Loan 1989; Hageman and Young 1981) for a more complete discussion of polynomial acceleration techniques and Krylov subspace methods. The close relationship between CG and the Lanzcos method is discussed in (Golub and Van Loan 1989).

The *preconditioned conjugate gradient* (PCG) *method* can be viewed either as an acceleration technique for the basic iterative method (18) or as a CG applied to the preconditioned system $M^{-1}Ax = M^{-1}b$, where

the splitting matrix M of (18) is typically called a preconditioning matrix in this context. We adopt the second point of view in this subsection.

An instructive way of deriving PCG is to exploit its relationship to the minimization technique of the same name described in §7.5. To this end, assume that the basic iterative method (18) is *symmetrizable*. That is, there exists a real, nonsingular matrix W such that $S = WM^{-1}AW^{-1}$ is a real, symmetric, positive-definite (SPD) matrix. Consider the quadratic functional $F(y) = \frac{1}{2}y^T Sy - y^T\hat{b}$, where $\hat{b} = WM^{-1}b$ and $y \in \mathbb{R}^n$. It is easy to show that the unique minimum of $F(y)$ is the solution $\hat{x}$ of $S\hat{x} = \hat{b}$. It follows immediately from the relations for S and $\hat{b}$ given above that $x = W^{-1}\hat{x}$ is the solution of both the preconditioned system $M^{-1}Ax = M^{-1}b$ and the original system $Ax = b$. If we take $M = W = I$, then $S = W(M^{-1}A)W^{-1} = A$ and, assuming A is a real SPD matrix, PCG reduces to the standard (unpreconditioned) CG method.

It is easy to show that, if both A and M are real SPD matrices, then $M^{-1}A$ is symmetrizable. Moreover, many important practical problems, such as the numerical solution of self-adjoint elliptic PDEs, give rise to matrices A that are real SPD. Furthermore, if A is a real SPD matrix, then the splitting matrix M associated with the RF, Jacobi and SSOR (with $\omega \in (0, 2)$) iterations is a real SPD matrix too, as is the M given by an incomplete Cholesky factorization, provided it exists. Hence, these basic iterative methods are symmetrizable in this case and so PCG can be used to accelerate their convergence. In contrast, the SOR iteration with the optimal ω is generally not symmetrizable. So PCG is not even applicable and more general Krylov subspace methods normally do not accelerate its convergence.

We assume throughout the rest of this subsection that A and M are real SPD matrices, because this case arises most frequently in practice and also because it simplifies the discussion below. Using this assumption and applying several mathematical identities, we get the computationally effective variant of PCG shown in Table 6. Note that W does not appear explicitly in this algorithm. Also note that, if we choose $M = I$, then $\tilde{r}_k = r_k$ and the PCG method reduces to the unpreconditioned CG method for $Ax = b$.

Many other mathematically equivalent forms of PCG exist. Moreover, as noted above, a more general form of PCG can be used if $M^{-1}A$ is symmetrizable, without either A or M being a real SPD matrix. For a discussion of the points, see an advanced text such as (Hageman and Young 1981).

Before considering the convergence of PCG, we introduce some notation. The *energy norm* of a vector $y \in \mathbb{R}^n$ with respect to a real SPD matrix B is $\|y\|_{B^{1/2}} = \sqrt{y^T By}$. The *Krylov subspace* of degree $k-1$ generated by a vector v and a matrix W is $\mathcal{K}_k(v, W) = \text{span}\{v, Wv, \ldots, W^{k-1}v\}$.

It is easy to show that the r_k that occurs in Table 6 is the residual $r_k = b - Ax_k$ associated with x_k for the system $Ax = b$ and that $\tilde{r}_k = M^{-1}r_k = M^{-1}b - M^{-1}Ax_k$ is the residual associated with x_k for the preconditioned system $M^{-1}Ax = M^{-1}b$. Let $e_k = x - x_k$ be the error associated with x_k. It can be shown that the x_k generated by PCG is a member of the shifted Krylov subspace $x_0 + \mathcal{K}_k(\tilde{r}_0, M^{-1}A) \equiv \{x_0 + v : v \in \mathcal{K}_k(\tilde{r}_0, M^{-1}A)\}$. Hence, PCG is in the broader class of Krylov subspace methods characterized by this property. Moreover, it can be shown that the x_k generated by PCG is the unique member of $x_0 + \mathcal{K}_k(\tilde{r}_0, M^{-1}A)$ that minimizes the energy norm of the error $\|e_k\|_{A^{1/2}} = \sqrt{e_k^T Ae_k}$ over all vectors of the form $e'_k = x - x'_k$, where x'_k is any other member of $x_0 + \mathcal{K}_k(\tilde{r}_0, M^{-1}A)$. Equivalently, $e_k = P_k^*(M^{-1}A)e_0$, where $P_k^*(z)$ is the polynomial that minimizes $\|P_k(M^{-1}A)e_0\|_{A^{1/2}}$ over all polynomials $P_k(z)$ of degree k

that satisfy $P_k(0) = 1$. This result is the basis of the characterization that PCG is the optimal polynomial acceleration scheme for the basic iterative method (18).

In passing, note that the iterate x_k generated by the basic iteration (18) is in the shifted Krylov subspace $x_0 + \mathcal{K}_k(\tilde{r}_0, M^{-1}A)$ also and that, by (21), the associated error satisfies $e_k = (I - M^{-1}A)^k e_0$. So (18) is a Krylov subspace method too and its error satisfies the polynomial relation described above with $P_k(z) = (1-z)^k$. Thus, the associated PCG method is guaranteed to accelerate the convergence of the basic iteration (18) — at least when the errors are measured in the energy norm.

The characterization of its error discussed above can be very useful in understanding the performance of PCG. For example, it can be used to prove the *finite termination property* of PCG. That is, if $M^{-1}A$ has m distinct eigenvalues, then $x_m = x$, the exact solution of $Ax = b$. Since $M^{-1}A$ has n eigenvalues, $m \leq n$ always and $m \ll n$ sometimes. We caution the reader, though, that the argument used to prove this property of PCG assumes exact arithmetic. In floating-point arithmetic, we rarely get $e_m = 0$, although we frequently get $e_{\tilde{m}}$ is small for some $\tilde{m}$ possibly a little larger than m.

The proof of the finite termination property can be extended easily to explain the rapid convergence of PCG when the eigenvalues of $M^{-1}A$ fall into a few small clusters. So a preconditioner M is good if the eigenvalues of $M^{-1}A$ are much more closely clustered than those of the unpreconditioned matrix A.

Because of the finite termination property, both CG and PCG can be considered direct methods. However, both are frequently used as iterative schemes, with the iteration terminated long before $e_m = 0$. Therefore, it is important to understand how the error decreases with k, the iteration count. To this end, first note that the characterization of the error ensures that $\|e_{k+1}\|_{A^{1/2}} \leq \|e_k\|_{A^{1/2}}$ with equality only if $e_k = 0$. That is, PCG is a *descent* method in the sense that some norm of the error decreases on every iteration. Not all iterative methods enjoy this useful property.

The characterization of the error can also be used to show that

$$\|e_k\|_{A^{1/2}} \leq 2 \left(\frac{\sqrt{\lambda_n/\lambda_1} - 1}{\sqrt{\lambda_n/\lambda_1} + 1} \right)^k \|e_0\|_{A^{1/2}} \tag{34}$$

where λ_n and λ_1 are the largest and smallest eigenvalues, respectively, of $M^{-1}A$. (Note λ_n and $\lambda_1 \in \mathbb{R}$ and $\lambda_n \geq \lambda_1 > 0$ since $M^{-1}A$ is symmetrizable.) It follows easily from the definition of the energy norm and (34) that

$$\|e_k\|_2 \leq 2\sqrt{\kappa_2(A)} \left(\frac{\sqrt{\lambda_n/\lambda_1} - 1}{\sqrt{\lambda_n/\lambda_1} + 1} \right)^k \|e_0\|_2 \tag{35}$$

where $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$ is the condition number of A in the 2-norm (see §2.8). Although (35) is generally not as tight as (34), it may be more relevant to the practitioner. It follows from either (34) or (35) that, in this context, a preconditioner M is good if λ_n/λ_1 is (much) closer to 1 than is the ratio of the largest to smallest eigenvalues of A.

Unlike many other iterative methods, such as SOR, PCG does not require an estimate of any parameters, although some stopping procedures for PCG require an estimate of the extreme eigenvalues of A or $M^{-1}A$. See an advanced text such as (Axelsson 1994; Golub and Van Loan 1989; Hageman and Young 1981) for

details.

4 Over-Determined and Under-Determined Linear Systems

A linear system $Ax = b$, with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ given and $x \in \mathbb{R}^n$ unknown, is called *over-determined* if $m > n$. Such a system typically has no solution. However, there is always an x that minimizes $\|b - Ax\|_2$. Such an x is called a *least squares* solution to the over-determined linear system $Ax = b$. Moreover, if $\text{rank}(A) = n$, then the least squares solution is unique.

A linear system $Ax = b$, as above, is called *under-determined* if $m < n$. Such a system typically has infinitely many solutions. If $Ax = b$ has a solution, then there is an x of minimum Euclidean norm, $\|x\|_2$, that satisfies $Ax = b$. Such an x is called a *least squares* solution to the under-determined linear system $Ax = b$. It can be shown that, if $rank(A) = m$, then the least squares solution is unique.

In the following subsections, we describe methods to compute the least squares solution of over-determined and under-determined linear systems, assuming that the matrix A has full rank, i.e., $\text{rank}(A) = \min(m, n)$. For the more general case of $\text{rank}(A) < \min(m, n)$, see an advanced text such as (Golub and Van Loan 1989).

4.1 The Normal Equations for Over-Determined Linear Systems

Let $Ax = b$ be an over-determined linear system with m equations in n unknowns and with $\text{rank}(A) = n$. The x that minimizes $\|b - Ax\|_2$ satisfies

$$\frac{\partial (b - Ax)^T (b - Ax)}{\partial x_j} = 0 \quad \text{for } j = 1, \ldots, n.$$

These relations can be rewritten in matrix form as $A^T(b - Ax) = 0$ or equivalently

$$A^T A x = A^T b \tag{36}$$

which is called the system of *normal equations* for the over-determined linear system $Ax = b$. It can be shown that the matrix $A^T A$ is symmetric and positive definite if $\text{rank}(A) = n$. So the system (36) has a unique solution, which is the least squares solution to $Ax = b$.

Note that the matrix $A^T A$ is $n \times n$. Computing the matrix $A^T A$ requires about $mn^2/2$ flops (floating-point operations), while computing $A^T b$ requires about mn flops, and solving (36) requires about $n^3/6$ flops, assuming the Cholesky factorization (see §2.6) is used.

The condition number of the matrix $A^T A$ is often large. More specifically, it can be shown that, if A is square and nonsingular, then the condition number of $A^T A$ is the square of the condition number of A (see §2.8). As a result, the approach described above of forming and solving the normal equations (36) often leads to a serious loss of accuracy. Therefore, in §4.4 and §4.7, we discuss more stable alternatives for solving least squares problems.

4.2 The Normal Equations for Under-Determined Linear Systems

Let $Ax = b$ be an under-determined linear system with m equations in n unknowns and with $\text{rank}(A) = m$. It can be shown that the least squares solution x to $Ax = b$ can be written in the form $A^T y$ for some $y \in \mathbb{R}^m$ which satisfies

$$AA^T y = b \tag{37}$$

This is a linear system of size $m \times m$, called the system of *normal equations* for the under-determined linear system $Ax = b$. It can be shown that the matrix AA^T is symmetric and positive definite if $\text{rank}(A) = m$. So the system (37) has a unique solution. The unique least squares solution to $Ax = b$ is $x = A^T y$.

Computing the matrix AA^T requires about $nm^2/2$ flops, while computing $A^T y$ requires about mn flops, and solving (37) requires about $m^3/6$ flops, assuming the Cholesky factorization (see §2.6) is used.

As is the case for over-determined systems, the method of forming and solving the normal equations (37) to compute the least squares solution to $Ax = b$ is numerically unstable in some cases. In §4.5 and §4.8, we discuss more stable alternatives.

4.3 Householder Transformations and the QR Factorization

An *orthogonal transformation* is a linear change of variables that preserves the length of vectors in the Euclidean norm. Examples are a rotation about an axis or a reflection across a plane. The following is an example of an orthogonal transformation $y = (y_1, y_2)$ to $x = (x_1, x_2)$:

$$\begin{aligned} x_1 &= 0.6y_1 + 0.8y_2 \\ x_2 &= 0.8y_1 - 0.6y_2 \end{aligned}$$

It is easy to see that $\|x\|_2 = \|y\|_2$.

An *orthogonal matrix* is a $m \times n$ matrix Q with the property $Q^T Q = I$. Note that, if (and only if) Q is square, i.e., $m = n$, the relation $Q^T Q = I$ is equivalent to $QQ^T = I$ and so $Q^T = Q^{-1}$. An orthogonal transformation of y to x can be written as $x = Qy$, where Q is an orthogonal matrix. In the above example, the corresponding orthogonal matrix is

$$Q = \begin{pmatrix} 0.6 & 0.8 \\ 0.8 & -0.6 \end{pmatrix}$$

If Q is an orthogonal matrix, then

$$\|x\|_2^2 = \|Qy\|_2^2 = (Qy)^T (Qy) = y^T (Q^T Q) y = y^T y = \|y\|_2^2,$$

whence $\|x\|_2 = \|y\|_2$. That is, orthogonal transformations preserve the Euclidean norm of a vector. This property is exploited in the methods described below to solve least squares problems. Moreover, the numerical stability of these schemes is due at least in part to the related observation that orthogonal transformations

do not magnify rounding error.

A *Householder transformation* or *Householder reflection* is an orthogonal matrix of the form $H = I - 2ww^T$, where $\|w\|_2 = 1$. Note that, when H is 2×2, the effect of H on a vector x is equivalent to reflecting the vector x across the plane perpendicular to w and passing through the origin of x. A Householder reflection can be used to transform a non-zero vector into one containing mainly zeros.

An $m \times n$ matrix $R = [r_{ij}]$ is *right triangular* if $r_{ij} = 0$ for $i > j$. Note that if (and only if) R is square, i.e., $m = n$, the terms right triangular and upper triangular are equivalent.

Let A be an $m \times n$ matrix with $m \geq n$. The QR factorization of A expresses A as the product of an $m \times m$ orthogonal matrix Q and an $m \times n$ right triangular matrix R. It can be computed by a sequence $H_1, H_2, \ldots, H_n$ of Householder transformations to reduce A to right triangular form R. More specifically, this variant of QR factorization proceeds in n steps (or $n-1$ if $n = m$). Starting with $A_0 = A$, at step k for $k = 1, \ldots, n$, H_k is applied to the partially processed matrix A_{k-1} to zero components $k+1$ to m of column k of A_{k-1}. $Q = H_1 H_2 \cdots H_n$ and $R = A_n$, the last matrix to be computed. For the details of the QR factorization algorithm using Householder transformations or other elementary orthogonal transformations, see (Hager 1988; Golub and Van Loan 1989).

4.4 Using the QR Factorization to Solve Over-Determined Linear Systems

Assume A is an $m \times n$ matrix, with $m \geq n$ and $\text{rank}(A) = n$. Let $Ax = b$ be the linear system to be solved (in the least squares sense, if $m > n$). Let $A = QR$ be the QR factorization of A, where Q is an $m \times m$ orthogonal matrix and R is an $m \times n$ right triangular matrix. Note that

$$\|Ax - b\|_2 = \|QRx - b\|_2 = \|Q(Rx - Q^T b)\|_2 = \|Rx - Q^T b\|_2.$$

Therefore, we can use the QR factorization of A to reduce the problem of solving $Ax = b$ to that of solving $Rx = Q^T b$, a much simpler task. We solve the latter by first computing $y = Q^T b$. Let $\hat{y}$ be the vector consisting of the first n components of y and $\hat{R}$ the upper triangular matrix consisting of the first n rows of R. Now solve $\hat{R}x = \hat{y}$ by back substitution (see §2.2). Then x is the least squares solution to both $Rx = Q^T b$ and $Ax = b$.

It can be shown that the QR factorization algorithm applied to an $n \times n$ linear system requires about $2n^3/3$ flops, which is about twice as many as the LU factorization needs. However, QR is a more stable method than LU and it requires no pivoting. The QR factorization algorithm applied to an $m \times n$ linear system requires about twice as many flops as forming and solving the normal equations. However, QR is a more stable method than solving the normal equations.

4.5 Using the QR Factorization to Solve Under-Determined Linear Systems

Assume A is an $m \times n$ matrix, with $m < n$ and $\text{rank}(A) = m$. Let $Ax = b$ be the linear system to be solved (in the least squares sense). Obtain the QR factorization of A^T by the QR factorization algorithm: $A^T = QR$, where Q is an $n \times n$ orthogonal matrix and R is an $n \times m$ right triangular matrix. Let $\hat{R}$ be

the upper triangular matrix consisting of the first m rows of R. Solve $\hat{R}^T\hat{y} = b$ by back substitution (see §2.2) and let $y = (\hat{y}^T, 0, \ldots, 0)^T \in \mathbb{R}^n$. Note that y is the vector of minimal Euclidean norm that satisfies $R^T y = b$. Finally compute $x = Qy$ and note that x is the vector of minimal Euclidean norm that satisfies $R^T Q^T x = b$ or equivalently $Ax = b$. That is, x is the least squares solution to $Ax = b$.

4.6 The Gram-Schmidt Orthogonalization Algorithm

The *Gram-Schmidt orthogonalization* algorithm is an alternative to QR. The modified version of the algorithm presented below is approximately twice as fast as QR and more stable than solving the normal equations.

Assume $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ and A having n linearly independent columns $\{a_j\}_{j=1}^n$. The Gram-Schmidt algorithm applied to A generates an $m \times n$ orthogonal matrix Q and an $n \times n$ upper triangular matrix R satisfying $A = QR$. In Table 7, we present a stable version of the algorithm, often called the *modified Gram-Schmidt algorithm.*

4.7 Using Gram-Schmidt to Solve Over-Determined Linear Systems

Assume A is an $m \times n$ matrix, with $m \geq n$ and $\text{rank}(A) = n$. Let $Ax = b$ be the linear system to be solved (in the least squares sense, if non-square). The method for solving $Ax = b$ using the Gram-Schmidt algorithm is similar to that described in §4.4 for the QR factorization.

First compute the QR factorization of A by the Gram-Schmidt algorithm, i.e., $A = QR$, where Q is an $m \times n$ orthogonal matrix and R is an $n \times n$ upper triangular matrix. Next compute $y = Q^T b$ and solve $Rx = y$ by back substitution (see §2.2). Then x is the least squares solution to $Ax = b$.

It can be shown that, the Gram-Schmidt algorithm applied to an $n \times n$ linear system requires about $n^3/3$ flops, which is about the same number of arithmetic operations as the LU factorization algorithm needs and about half of what the QR factorization algorithm requires. Moreover, the modified Gram-Schmidt algorithm, as presented in Table 7, is relatively stable. The Gram-Schmidt algorithm applied to an $m \times n$ linear system requires about the same number of flops as that needed to form and to solve the normal equations, but the Gram-Schmidt algorithm is more stable.

4.8 Using Gram-Schmidt to Solve Under-Determined Linear Systems

Assume A is an $m \times n$ matrix, with $m < n$ and $\text{rank}(A) = m$. Let $Ax = b$ be the linear system to be solved (in the least squares sense). The method for solving $Ax = b$ using the Gram-Schmidt algorithm is similar to that described in §4.5 for the QR factorization.

First compute the QR factorization of A^T by the Gram-Schmidt algorithm: $A^T = QR$, where Q is an $n \times m$ orthogonal matrix and R an $m \times m$ upper triangular matrix. Next solve $R^T y = b$ by back substitution (see §2.2) and set $x = Qy$. Then x is the least squares solution to $Ax = b$.

5 Eigenvalues and Eigenvectors of Matrices

Given an $n \times n$ matrix A, $\lambda \in \mathbb{C}$ and $x \in \mathbb{C}^n$ satisfying $Ax = \lambda x$, λ is called an *eigenvalue* of A and x is called an *eigenvector* of A.

The relation $Ax = \lambda x$ can be rewritten as $(A - \lambda I)x = 0$, emphasizing that λ is an eigenvalue of A if and only if $A - \lambda I$ is singular and that an eigenvector x is a non-trivial solution to $(A - \lambda I)x = 0$. It also follows that the eigenvalues of A are the roots of $\det(A - \lambda I) = 0$, the *characteristic equation* of A. The polynomial $p(\lambda) = \det(A - \lambda I)$ of degree n is called the *characteristic polynomial* of A and plays an important role in the theory of eigenvalues.

An $n \times n$ matrix A has precisely n eigenvalues, not necessarily distinct. It also has at least one eigenvector for each distinct eigenvalue. Note also that, if x is an eigenvector of A, then so is any (scalar) multiple of x and the corresponding eigenvalue is the same. We often choose an eigenvector of norm one in some vector norm, often the Euclidean norm, as the representative.

Two matrices A and B are *similar* if $A = W^{-1}BW$ for some non-singular matrix W. The matrix W is often referred to as *similarity transformation*. It is easy to see that, if $Ax = \lambda x$, then $B(Wx) = \lambda(Wx)$. Thus, similar matrices have the same eigenvalues and their eigenvectors are related by the similarity transformation W. Similarity transformations are often used in numerical methods for eigenvalues and eigenvectors to transform a matrix A into another one B that has the same eigenvalues and related eigenvectors, but the eigenvalues and eigenvectors of B are in some sense easier to compute than those of A.

Eigenvalues play a major role in the study of convergence of iterative methods (see §3). Eigenvalues and eigenvectors are also of great importance in understanding the stability and other fundamental properties of many physical systems.

The matrix A is often large, sparse and symmetric. These properties can be exploited to great advantage in numerical schemes for calculating the eigenvalues and eigenvectors of A.

A common approach to calculate the eigenvalues (and possibly the eigenvectors) of a matrix A consists of two stages. First, the matrix A is transformed to a similar but simpler matrix B, usually tridiagonal, if A is symmetric (or Hermitian), or Hessenberg, if A is non-symmetric (or non-Hermitian). Then, the eigenvalues (and possibly the eigenvectors) of B are calculated. An exception to this approach is the power method (see §5.1).

A standard procedure for computing the eigenvectors of a matrix A is to calculate the eigenvalues first then use them to compute the eigenvectors by inverse iteration (see §5.4). Again, an exception to this approach is the power method (see §5.1).

Before describing numerical methods for the eigenvalue problem, we comment briefly on the sensitivity of the eigenvalues and eigenvectors to perturbations in the matrix A, since a backward error analysis can often show that the computed eigenvalues and eigenvectors are the exact eigenvalues and eigenvectors of a slightly perturbed matrix $\hat{A} = A + E$, where E is usually small relative to A. In general, if A is symmetric, its eigenvalues are well-conditioned with respect to small perturbations E. That is, the eigenvalues of $\hat{A}$ and A are very close. This, though, is not always true of the eigenvectors of A, particularly if the associated eigenvalue is a multiple eigenvalue or close to another eigenvalue of A. If A is non-symmetric, then both its

eigenvalues and eigenvectors may be poorly conditioned with respect to small perturbations E. Therefore, the user of a computer package for calculating the eigenvalues of a matrix should be cautious about the accuracy of the numerical results. For a further discussion of the conditioning of the eigenvalue problem, see (Wilkinson 1965).

5.1 The Power Method

The power method is used to calculate the eigenvalue of largest magnitude of a matrix A and the associated eigenvector. Since matrix-vector products are the dominant computational work required by the power method, this scheme can exploit the sparsity of the matrix to great advantage.

Let λ be the eigenvalue of A of largest magnitude and let x be an associated eigenvector. Also let z_0 be an initial guess for some multiple of x. The power method, shown in Table 8, is an iterative scheme that generates a sequence of approximations $z_1, z_2, \ldots$ to some multiple of x and another sequence of approximations $\mu_1, \mu_2, \ldots$ to λ. In the scheme shown in Table 8, z_k is normalized so that the sequence $z_1, z_2, \ldots$ converges to an eigenvector x of A satisfying $\|x\|_\infty = 1$. Normalizations of this sort are frequently used in eigenvector calculations.

The power method is guaranteed to converge if A has a single eigenvalue λ of largest magnitude. The rate of convergence depends on $|\lambda_2|/|\lambda|$, where λ_2 is the eigenvalue of A of next largest magnitude. With some modifications the power method can be used when A has more than one eigenvalue of largest magnitude.

After the absolutely largest eigenvalue of A has been calculated, the power method can be applied to an appropriately deflated matrix to calculate the next largest eigenvalue and the associated eigenvector of A, and so on. However, this approach is inefficient if all or many eigenvalues are needed. The next sections describe more general purpose methods for eigenvalue and eigenvector computations. For an introduction to the power method, see (Atkinson 1989). For further reading, see (Golub and Van Loan 1989; Wilkinson 1965).

5.2 The QR Method

The QR method is widely used to calculate all the eigenvalues of a matrix A. It employs the QR factorization algorithm presented briefly in §4.4. Here, we recall that, given an $n \times n$ matrix A, there is a factorization $A = QR$, where R an $n \times n$ upper triangular matrix and Q an $n \times n$ orthogonal matrix.

The QR method, shown in Table 9, is an iterative scheme to compute the eigenvalues of A. It proceeds by generating a sequence $A_1, A_2, \ldots$ of matrices, all of which are similar to each other and to the starting matrix $A_0 = A$. The sequence converges either to a triangular matrix with the eigenvalues of A on its diagonal or to an almost triangular matrix from which the eigenvalues can be calculated easily.

For a real, nonsingular matrix A with no two (or more) eigenvalues of the same magnitude, the QR method is guaranteed to converge. The iterates A_k converge to an upper triangular matrix with the eigenvalues of A on its diagonal. If A is symmetric, the iterates converge to a diagonal matrix. The rate of convergence depends on $\max\{|\lambda_{i+1}/\lambda_i| : i = 1, \ldots, n-1\}$, where $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$.

If two or more eigenvalues of A have the same magnitude, the sequence $A_1, A_2, \ldots$ may not converge to an upper triangular matrix. If A is symmetric, the sequence converges to a block diagonal matrix, with blocks of order 1 or 2, from which the eigenvalues of A can be calculated easily. If A is non-symmetric, the problem is more complicated. See (Wilkinson 1965; Parlett 1968) for details.

In many cases, the QR method is combined with a technique known as *shifting* to accelerate convergence. A discussion of this procedure can be found in (Atkinson 1989; Golub and Van Loan 1989; Parlett 1980; Wilkinson 1965).

The QR method for eigenvalues requires the computation of the QR factorization of a matrix and a matrix-matrix multiplication at each iteration. For a large matrix A, this is an expensive computation, making the method inefficient. To improve its efficiency, the matrix A is normally preprocessed, reducing it to a simpler form. If A is symmetric, it is reduced to a similar tridiagonal matrix, as described in §5.3. If A is non-symmetric, it is reduced to a similar upper Hessenberg matrix by a comparable algorithm. It can be shown that, if $A_0 = A$ is in Hessenberg or tridiagonal form, then all the iterates A_k generated by the QR method will also be in Hessenberg or tridiagonal form, respectively. By using appropriate implementation techniques, the QR factorization and the matrix-matrix products applied to matrices with these special structures require fewer operations than would be needed for arbitrary dense matrices.

With the modifications discussed above, the QR method is an efficient, general purpose scheme for calculating the eigenvalues of a dense matrix. The eigenvectors can also be calculated if all the similarity transformations employed in the QR process are stored. Alternatively, the eigenvectors can be calculated by inverse iterations, as described in §5.4.

5.3 Transforming a Symmetric Matrix to Tridiagonal Form

As noted above, the eigenvalues of a symmetric (or Hermitian) matrix A are often calculated by first transforming A into a similar tridiagonal matrix T. Householder transformations (see §4.3) are usually employed to perform this task.

The algorithm to obtain the matrix T, given A, resembles the QR factorization algorithm described briefly in §4.3. It proceeds in $n-2$ steps and generates a sequence $H_1, H_2, \ldots, H_{n-2}$ of Householder transformations to reduce A to tridiagonal form T. Starting with $A^{(0)} = A$, at step k for $k = 1, \ldots, n-2$, we form $A_k = H_k A_{k-1} H_k$, where H_k is chosen to zero components $k+2$ to n of both row k and column k of A_{k-1}. T is A_{n-2}, the last matrix computed by this process. Note that $T = H_{n-2} \cdots H_1 A H_1 \cdots H_{n-2}$ is similar to A because each H_k is symmetric and orthogonal.

It can be shown that the reduction to tridiagonal form by Householder transformations is a stable computation in the sense that the computed T is the exact T for a slightly perturbed matrix $\hat{A} = A + E$, where E is usually small relative to A. As a result, it can be shown that the eigenvalues of A and T differ very little. For a brief introduction to tridiagonal reduction, see (Atkinson 1989). For further reading, see (Golub and Van Loan 1989; Wilkinson 1965). For other methods to reduce A to tridiagonal form, such as planar rotation orthogonal matrices, see (Golub and Van Loan 1989).

Similar schemes can be used to reduce a non-symmetric (or non-Hermitian) matrix to Hessenberg form.

5.4 Inverse Iteration

Inverse iteration is the standard method to calculate the eigenvectors of a matrix A, once its eigenvalues have been calculated. This scheme can be viewed as the power method (see §5.1) applied to the matrix $(A - \tilde{\lambda}I)^{-1}$ instead of A, where $\tilde{\lambda}$ is an (approximate) eigenvalue of A.

To see how inverse iteration works, let $\tilde{\lambda}$ be an approximation to a simple eigenvalue λ of A and let x be the associated eigenvector. Also let z_0 be an initial guess to some multiple of x. Inverse iteration, shown in Table 10, is an iterative method that generates a sequence of approximations $z_1, z_2, \ldots$ to some multiple of x. In the scheme shown in Table 10, z_k is normalized so that the sequence $z_1, z_2, \ldots$ converges to an eigenvector x of A satisfying $\|x\|_\infty = 1$. As noted already in §5.1, normalizations of this sort are frequently used in eigenvector calculations.

Note that, if $\tilde{\lambda} = \lambda$, then $A - \tilde{\lambda}I$ is singular. Moreover, if $\tilde{\lambda} \approx \lambda$, then $A - \tilde{\lambda}I$ is "nearly" singular. As a result, we can expect the system $(A - \tilde{\lambda}I)w_k = z_{k-1}$ to be very poorly conditioned (see §2.8). This though is not a problem in this context, since any large perturbation in the solution of the ill-conditioned system is in the direction of the desired eigenvector x.

The sequence of approximate eigenvectors $z_1, z_2, \ldots$ is typically calculated by first performing an LU decomposition of $A - \tilde{\lambda}I$, often with pivoting, before the start of the iteration and then using the same LU factorization to perform a forward elimination followed by a back substitution to solve $(A - \tilde{\lambda}I)w_k = z_{k-1}$ for $k = 1, 2, \ldots$ (see §2). We emphasize that only one LU factorization of $A - \tilde{\lambda}I$ is needed to solve all $(A - \tilde{\lambda}I)w_k = z_{k-1}$, $k = 1, 2, \ldots$

For a brief discussion of the inverse iteration, including its stability and rate of convergence, see (Atkinson 1989). For further reading, see (Wilkinson 1965).

5.5 Other Methods

Another way to calculate the eigenvalues of A is to compute the roots of the characteristic polynomial $p(\lambda) = \det(A - \lambda I)$. The techniques discussed in §6.9 can be used for this purpose. However, this approach is often less stable that the techniques described above and it is usually not more efficient. Therefore, it is not normally recommended.

An obvious method for calculating an eigenvector x, once the corresponding eigenvalue λ is known, is to solve the system $(A - \lambda I)x = 0$. Since this system is singular, one approach is to delete one equation from $(A - \lambda I)x = 0$ and replace it by another linear constraint, such as $x_j = 1$ for some component j of x. However, it can be shown (see (Wilkinson 1965)) that this method is not always stable and can lead to very poor numerical results in some cases. Thus, it is not recommended either.

Several other methods for calculating the eigenvalues and eigenvectors of a matrix have been omitted from our discussion due to space limitations. We should, though, at least mention one of them, the Jacobi method, a simple, rapidly convergent iterative scheme, applicable to symmetric matrices, including sparse ones.

The eigenvalue problem for large sparse matrices is a very active area of research. Although the QR

method described in §5.2 is effective for small to medium sized dense matrices, it is computationally very expensive for large matrices, in part because it does not exploit sparsity effectively. The Lanczos and Arnoldi methods are much better suited for large sparse problems. For a discussion of these methods, see (Cullum and Willoughby 1985; Parlett 1980; Saad 1992; Scott 1981).

6 Nonlinear Algebraic Equations and Systems

Consider a nonlinear algebraic equation or system $f(x) = 0$, where $f : \mathbb{R}^n \to \mathbb{R}^n$, $n \geq 1$. A *root* of $f(x)$ is a value $\alpha \in \mathbb{R}^n$ satisfying $f(\alpha) = 0$. A nonlinear function may have one, many or no roots.

Most numerical methods for computing approximations to roots of a nonlinear equation or system are *iterative* in nature. That is, the scheme starts with some initial guess x_0 and computes new successive approximation x_k, $k = 1, 2, \ldots$, by some formula until a *stopping criterion* such as

$$\begin{aligned} \|f(x_k)\| &\leq \epsilon, \\ \|f(x_k)\| &\leq \epsilon \|f(x_0)\|, \\ \|x_k - x_{k-1}\| &\leq \epsilon, \\ \|x_k - x_{k-1}\| &\leq \epsilon \|x_k\|, \quad \text{or} \\ k &> \text{maxit} \end{aligned}$$

is satisfied, where ϵ is the error tolerance and maxit is the maximum number of iterations allowed.

6.1 Fixed-Point Iteration

A *fixed point* of a function $g(x)$ is a value α satisfying $\alpha = g(\alpha)$. A *fixed-point iteration* is a scheme of the form $x_k = g(x_{k-1})$ that uses the most recent approximation x_{k-1} to the fixed-point α to compute a new approximation x_k to α. In this context, the function g is also called the *iteration function*.

One reason for studying fixed-point iterations is that, given a function $f(x)$, it is easy to find another function $g(x)$ such that α is a root of $f(x)$ if and only if it is a fixed-point of $g(x)$. For example, take $g(x) = x - f(x)$. Many root-finding methods can be viewed as fixed-point iterations.

Given an iteration function g, a fixed-point scheme starts with an initial guess x_0 and proceeds with the iteration as follows:

```
for k = 1, 2, ... do
    x_k = g(x_{k-1})
    test stopping criterion
end
```

6.2 Newton's Method for Nonlinear Equations

We consider scalar equations (i.e., $n = 1$) first, and extend the results to systems of equations (i.e., $n > 1$) in §6.7.

Newton's method is a fixed-point iteration based on the iteration function $g(x) = x - f(x)/f'(x)$, where $f'(x)$ is the first derivative of f. More specifically, the new approximation x_k to the root α of f is computed by the formula

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

which uses the previous approximation x_{k-1} to α. Geometrically, Newton's method approximates the nonlinear function $f(x)$ by its tangent (a straight line) at the current approximation x_{k-1} and takes x_k to be the intersection of the tangent with the x axis. That is, x_k is the root of the local linear approximation to $f(x)$. Newton's method is applicable if and only if f is differentiable and f' is non-zero at the point of approximation.

6.3 The Secant Method

The secant method is applicable to scalar equations only and is not a fixed-point iteration. The new approximation x_k to the root α of f is computed using two previous approximations x_{k-1} and x_{k-2} by the formula

$$x_k = x_{k-1} - f(x_{k-1})\frac{x_{k-1} - x_{k-2}}{f(x_{k-1}) - f(x_{k-2})}.$$

The secant method can be viewed as a variant of Newton's method in which $f'(x_{k-1})$ is approximated by $[f(x_{k-1}) - f(x_{k-2})]/[x_{k-1} - x_{k-2}]$. Geometrically, the secant method approximates the nonlinear function $f(x)$ by the chord subtending the graph of f at the two most recently computed points of approximation x_{k-1} and x_{k-2} and takes x_k to be the intersection of the chord with the x axis. That is, x_k is the root of the local linear approximation to $f(x)$. The secant method is applicable if and only if $f(x)$ is continuous and takes different values at x_{k-1} and x_{k-2}. To start, the secant method requires initial guesses for x_0 and x_1. These are usually chosen close to each other and must satisfy $f(x_0) \neq f(x_1)$.

6.4 The Bisection and Regula Falsi Methods

The bisection method is not a fixed-point iteration. It is applicable to a scalar equation $f(x) = 0$ if and only if $f(x)$ is continuous and there are two points L and R for which $f(L) \cdot f(R) \leq 0$. These conditions guarantee the existence of at least one root of $f(x)$ in the interval $[L, R]$. Without loss of generality, let $L < R$. To start, the bisection method approximates the root by the mid-point $M = (L + R)/2$ of $[L, R]$ and halves the interval at each iteration as follows.

forever do
  $M = (L + R)/2$
  if $f(L) \cdot f(M) \leq 0$ then $R = M$
    else $L = M$

test stopping criterion

end

Note that this iteration maintains the property $f(L) \cdot f(R) \leq 0$, as L and R are changed. So, when the algorithm terminates, a root of f is guaranteed to be in $[L, R]$. $M = (L + R)/2$ is often taken as the approximation to the root.

Several root-finding methods are similar to bisection. For example, *regula falsi* chooses

$$M = \frac{f(R)L - f(L)R}{f(R) - f(L)} \tag{38}$$

but is otherwise the same as bisection. Note that the M computed from (38) is the intersection of the chord subtending the graph of $f(x)$ at L and R with the x-axis and so is guaranteed to lie in $[L, R]$, since the property $f(L) \cdot f(R) \leq 0$ is maintained throughout the iteration even though L and R are changed.

6.5 Convergence

Iterative methods for nonlinear equations can be guaranteed to converge under certain conditions, although they may diverge in some cases.

The bisection method converges whenever it is applicable, but, if $f(x)$ has more than one root in the interval of application, there is no guarantee which of the roots the method will converge to.

The convergence of a fixed-point iteration depends critically on the properties of the iteration function $g(x)$. If g is smooth in an interval containing a fixed-point α and $|g'(\alpha)| < 1$, then there is an $m \in [0, 1)$ and a neighbourhood I around the fixed-point α in which $|g'(x)| \leq m < 1$. In this case, the fixed-point iteration $x_k = g(x_{k-1})$ converges to α if $x_0 \in I$. To give an intuitive understanding why this is so, we assume $x_0, \ldots, x_{k-1} \in I$ and note that

$$x_k - \alpha = g(x_{k-1}) - g(\alpha) = g'(\xi_k)(x_{k-1} - \alpha)$$

where we have used $x_k = g(x_{k-1})$, $\alpha = g(\alpha)$ and, by the mean value theorem, ξ_k is some point in I between x_{k-1} and α. Thus, $|g'(\xi_k)| \leq m < 1$ and so $|x_k - \alpha| \leq m|x_{k-1} - \alpha| \leq m^k|x_0 - \alpha|$, whence $x_k \in I$ too and $x_k \to \alpha$ as $k \to \infty$.

A more formal statement of this theorem and other similar results giving sufficient conditions for the convergence of fixed-point iterations can be found in many introductory numerical methods textbooks. See for example (Conte and de Boor 1980; Dahlquist and Björck 1974; Johnson and Riess 1982; Stoer and Bulirsch 1980).

Newton's method converges if the conditions for convergence of a fixed-point iteration are met. (For Newton's method, the iteration function is $g(x) = x - f(x)/f'(x)$.) However, it can be shown that Newton's method converges quadratically (see §6.6) to the root α of f if the initial guess x_0 is chosen sufficiently close to α, f is smooth and $f'(x) \neq 0$ close to α. It can also be shown that Newton's method converges from any starting guess in some cases. A more formal statement of these and other similar results can be found in

many introductory numerical methods textbooks. See for example (Conte and de Boor 1980; Dahlquist and Björck 1974; Johnson and Riess 1982; Stoer and Bulirsch 1980). For a deeper discussion of this topic, see (Dennis and Schnabel 1983).

6.6 Rate of Convergence

The rate of convergence of a sequence $x_1, x_2, \ldots$ to α is the largest number $p \geq 1$ satisfying

$$\|x_{k+1} - \alpha\| \leq C \|x_k - \alpha\|^p \quad \text{as } k \to \infty$$

for some constant $C > 0$. If $p = 1$, we also require that $C < 1$. The larger the value of p the faster the convergence, at least asymptotically. Between two converging sequences with the same rate p, the faster is the one with the smaller C.

A fixed-point iteration with iteration function g converges at a rate p with $C = |g^{(p)}(\alpha)|/p!$ if $g \in \mathcal{C}^p$, $g^{(i)}(\alpha) = 0$, for $i = 0, 1, 2, \ldots, p-1$, and $g^{(p)}(\alpha) \neq 0$, where $g^{(i)}(x)$ is the i^{th} derivative of $g(x)$.

Thus, Newton's method usually converges quadratically, i.e., $p = 2$ with $C = |g''(\alpha)|/2$, where $g(x) = x - f(x)/f'(x)$. If $f'(\alpha) = 0$, Newton's method typically converges linearly. If $f'(\alpha) \neq 0$ and $f''(\alpha) = 0$, Newton's method converges at least cubicly, i.e., $p \geq 3$.

The secant method converges at a superlinear rate of $p = (1 + \sqrt{5})/2 \approx 1.618$, i.e., faster than linear but slower than quadratic.

Bisection converges linearly, i.e., $p = 1$ and $C = 1/2$,

6.7 Newton's Method for Systems of Nonlinear Equations

Newton's method for a scalar nonlinear equation (see §6.2) can be extended to a system of nonlinear equations with $f : \mathbb{R}^n \to \mathbb{R}^n$, for $n > 1$. In this case, the new iterate x_k is computed by

$$x_k = x_{k-1} - [J(x_{k-1})]^{-1} f(x_{k-1})$$

where $J(x)$ is the *Jacobian* of f, an $n \times n$ matrix with its (i, j) entry equal to $\partial f_i(x)/\partial x_j$. To implement this scheme effectively, the matrix-vector product $z = [J(x_{k-1})]^{-1} f(x_{k-1})$ should NOT be calculated by first computing the inverse of the Jacobian and then performing the matrix-vector multiplication, but rather by first solving the linear system $[J(x_{k-1})]z_k = f(x_{k-1})$ by Gaussian elimination (see §2.1) or some other effective method for solving linear equations (see §2 and §3) and then setting $x_k = x_{k-1} - z_k$.

6.8 Modifications and Alternatives to Newton's Method

Solving the linear system $[J(x_{k-1})]z_k = f(x_{k-1})$ requires first evaluating all the partial derivatives of all components of f at the point x_{k-1} and then solving the linear system by performing Gaussian elimination (see §2.1) or some other effective method for solving linear equations (see §2 and §3). In the unmodified Newton's method, this procedure is repeated at every iteration, requiring $O(n^3)$ flops (floating-point operations) if

Gaussian elimination is used. There exist variants of Newton's method which reduce the computational load per iteration significantly. Even though these schemes typically converge more slowly, they often dramatically reduce the cost of solving a nonlinear system, particularly if $n \gg 1$.

The *chord Newton* method, often called the *simplified Newton method*, holds $J(x_{k-1})$ fixed for several steps, thus avoiding many Jacobian evaluations and LU factorizations. However, it still requires one f evaluation and both a forward elimination and a back substitution (see §2.4 and §2.2, respectively) at each iteration.

Some other variants approximate the Jacobian by matrices that are easier to compute and simpler to solve. For example, the Jacobian may be approximated by its diagonal, giving rise to a *Jacobi-like Newton's* iteration, or by its lower triangular part, giving rise to a *Gauss-Seidel-like Newton's* scheme. (See §3 for a discussion of Jacobi and Gauss-Seidel iterations for linear equations.)

Quasi-Newton schemes, which avoid the computation of partial derivatives, are alternatives to Newton's method. Some quasi-Newton schemes approximate partial derivatives by finite differences. For example,

$$[J(x)]_{ij} = \frac{\partial f_i}{\partial x_j}(x) \approx \frac{f_i(x + \delta e_j) - f_i(x)}{\delta},$$

where δ is a small non-zero number and $e_j \in \mathbb{R}^n$ is the j^{th} unit vector with a 1 in component j and 0's in all other entries.

Possibly the best-known quasi-Newton scheme is *Broyden's* method. It does not require the computation of any partial derivatives nor the solution of any linear systems. Rather, it uses one evaluation only of f and a matrix-vector multiply, requiring $O(n^2)$ flops, per iteration. Starting with an initial guess for the inverse of the Jacobian, $J(x_0)$, it updates its approximation to the inverse Jacobian at every iteration. Broyden's method can be viewed as an extension of the secant method to $n > 1$ dimensions. For a brief description of the algorithm, see (Hager 1988).

6.9 Polynomial Equations

Polynomial equations are a special case of nonlinear equations. A polynomial of degree k has exactly k roots, counting multiplicity. One may wish to compute all roots of a polynomial or only a few select ones. The methods for nonlinear equations described above can be used in either case, although more effective schemes exist for this special class of problems. The efficient evaluation of the polynomial and its derivative is discussed below in §6.10.

Deflation is often used to compute roots of polynomials. It starts by locating one root r_1 of $p(x)$ and then proceeds recursively to compute the roots of $\hat{p}(x)$, where $p(x) = (x - r_1)\hat{p}(x)$. Note that, by the fundamental theorem of algebra, given a root r_1 of $p(x)$, $\hat{p}(x)$ is uniquely defined and is a polynomial of degree $k - 1$. However, deflation may be unstable unless implemented carefully. See an introductory numerical methods textbook for details.

Localization techniques can be used to identify regions of the complex plane which contain zeros. Such techniques include the Lehmer-Schur method, Laguerre's method and methods based on Sturm sequences

(see (Householder 1970)). Localization can be very helpful, for example, when searching for a particular root of a polynomial or when implementing deflation, since in the latter case, the roots should be computed in increasing order of magnitude to ensure numerical stability.

The roots of a polynomial $p(x)$ can also be found by first forming the *companion matrix* A of the polynomial $p(x)$ — the eigenvalues $\{\lambda_i : i = 1, \ldots, n\}$ of A are the roots of $p(x)$ — and then finding all, or a select few, of the eigenvalues of A. See §5 for a discussion of the computation of eigenvalues and an introductory numerical methods book, such as (Hager 1988), for a further discussion of this root-finding technique.

6.10 Horner's Rule

Horner's rule, also called *nested multiplication* or *synthetic division*, is an efficient method to evaluate a polynomial and its derivative. Let $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ be the polynomial of interest, and α the point of evaluation. The following algorithm computes $p(\alpha)$ in z_0 and $p'(\alpha)$ in y_1 efficiently.

$z_n = a_n$
$y_n = a_n$
for $j = n - 1, \ldots, 1$ do
    $z_j = \alpha\, z_{j+1} + a_j$
    $y_j = \alpha\, y_{j+1} + z_j$
end
$z_0 = z_1 \alpha + a_0$

7 Unconstrained Optimization

The *optimization problem* is to find a value $x^* \in \mathbb{R}^n$ that either *minimizes* or *maximizes* a function $f : \mathbb{R}^n \to \mathbb{R}$. We consider only the minimization problem here, since maximizing $f(x)$ is equivalent to minimizing $-f(x)$.

Sometimes the minimizer must satisfy constraints such as $g_i(x) = 0$, $i = 1, \ldots, m_1$, or $h_j(x) \geq 0$, $j = 1, \ldots, m_2$, where g_i and $h_j : \mathbb{R}^n \to \mathbb{R}$. Thus, the general minimization problem can be written as

$$\min_{x \in \mathbb{R}^n} f(x)$$

subject to

$$\begin{aligned} g_i(x) &= 0, \quad i = 1, \ldots, m_1 \\ h_j(x) &\geq 0, \quad j = 1, \ldots, m_2 \end{aligned}$$

If any of the functions f, g_i or h_j are nonlinear, then the minimization problem is *nonlinear*; otherwise, it is called a *linear programming* problem. If there are no constraints, the minimization problem is called

unconstrained; otherwise, it is called *constrained*.

In this section, we present numerical methods for nonlinear unconstrained minimization problems (NLUMPs) only. For a more detailed discussion of these schemes, see an advanced text such as (Dennis and Schnabel 1983).

7.1 Some Definitions and Properties

Let $x = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. The *gradient* ∇f of the function f is the vector of the n first partial derivatives of f:

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)^T .$$

Note $\nabla f(x) : \mathbb{R}^n \to \mathbb{R}^n$.

A point x^* is a *critical point* of f if $\nabla f(x^*) = 0$. A point x^* is a *global minimum* of f if $f(x^*) \leq f(x)$ for all $x \in \mathbb{R}^n$. A point x^* is a *local minimum* of f if $f(x^*) \leq f(x)$ for all x in a neighbourhood of x^*. If x^* is a local minimum of f, then it is also a critical point of f, assuming $\nabla f(x^*)$ exists, but the converse is not necessarily true.

The *Hessian* $\nabla^2 f(x) = H(x)$ of f is an $n \times n$ matrix with entries

$$H_{ij}(x) = \frac{\partial^2 f}{\partial x_i \partial x_j}(x).$$

If f is smooth, then

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}$$

and so the Hessian is symmetric. A critical point x^* is a local minimum of f if $H(x^*)$ is symmetric positive-definite.

Numerical methods for solving NLUMPs are all iterative in character. Some try to compute roots of the gradient, i.e., critical points of f, which are also local minima. These methods are related to schemes used to solve nonlinear equations described in §6.

Minimization methods can be classified into three main categories:

- *direct search methods*, which make use of function values only,
- *gradient methods*, which make use of function and derivative values, and
- *Hessian methods*, which make use of function, derivative and second derivative values.

Methods from each class are discussed below.

7.2 The Fibonacci and Golden-Section Search Methods

The Fibonacci and golden-section search methods belong to the class of direct search methods. They are similar to the bisection method for nonlinear equations described in §6.4, although important differences exist. They are used in one-dimensional minimization only.

The Fibonacci and golden-section search methods are applicable if $f(x)$ is continuous and unimodal in the interval of interest $[a, b]$, where by *unimodal* we mean that $f(x)$ has exactly one local minimum $x^* \in [a, b]$ and $f(x)$ strictly decreases for $x \in [a, x^*]$ and strictly increases for $x \in [x^*, b]$.

The main idea behind these methods is the following. Let x_1 and x_2 be two points satisfying $a < x_1 < x_2 < b$. If $f(x_1) \geq f(x_2)$, then x^* must lie in $[x_1, b]$. On the other hand, if $f(x_1) \leq f(x_2)$ then x^* must lie in $[a, x_2]$. Thus, by evaluating f at a sequence of test points $x_1, x_2, \ldots, x_k$, we can successively reduce the length of the interval in which we know the local minimum lies, called the *interval of uncertainty*.

The Fibonacci and golden-section search methods differ only in the way the sequence of test points is chosen. Before describing the two methods, we give two more definitions.

The *coordinate* ν of a point x *relative to an interval* $[a, b]$ is $\nu = (x - a)/(b - a)$.

The *Fibonacci numbers* are defined by the initial condition $F_0 = F_1 = 1$ and the recurrence $F_k = F_{k-1} + F_{k-2}$ for $k \geq 2$.

The Fibonacci search method applied to a function f on an interval $[a, b]$ starts with two points x_k and x_{k-1} satisfying $(x_k - a)/(b - a) = F_{k-1}/F_k$ and $(x_{k-1} - a)/(b - a) = F_{k-2}/F_k = 1 - F_{k-1}/F_k$, whence $a < x_{k-1} < x_k < b$. It then computes the sequence $x_{k-2}, x_{k-3}, \ldots, x_1$ as follows. After evaluating $f(x_k)$ and $f(x_{k-1})$, the interval of uncertainty is either $[a, x_k]$ or $[x_{k-1}, b]$.

- If the interval of uncertainty is $[a, x_k]$, then x_{k-1} belongs to it and $(x_{k-1} - a)/(x_k - a) = F_{k-2}/F_{k-1}$. In this case, x_{k-2} is chosen to satisfy $(x_{k-2} - a)/(x_k - a) = F_{k-3}/F_{k-1}$. The method proceeds to the next iteration with the two points x_{k-1} and x_{k-2} in the interval $[a, x_k]$, with relative coordinates F_{k-2}/F_{k-1} and F_{k-3}/F_{k-1}, respectively. Note that $f(x_{k-1})$ is already computed, so $f(x_{k-2})$ only needs to be evaluated in the next iteration.

- If the interval of uncertainty is $[x_{k-1}, b]$, then x_k belongs to it and $(x_k - x_{k-1})/(b - x_{k-1}) = F_{k-3}/F_{k-1}$. In this case, x_{k-2} is chosen to satisfy $(x_{k-2} - x_{k-1})/(b - x_{k-1}) = F_{k-2}/F_{k-1}$. The method proceeds to the next iteration with the two points x_k and x_{k-2} in the interval $[x_{k-1}, b]$, with relative coordinates F_{k-3}/F_{k-1} and F_{k-2}/F_{k-1}, respectively. Note that $f(x_k)$ is already computed, so $f(x_{k-2})$ only needs to be evaluated in the next iteration.

Thus, at the start of the second iteration, the situation is similar to that at the start of the first iteration, except that the length of interval of uncertainty has been reduced. Therefore, the process described above can be repeated.

Before the last iteration, the interval of uncertainty is $[c, d]$ and x_1 is chosen to be $(c + d)/2 + \epsilon$, for some small positive number ϵ.

As noted above, the Fibonacci search method requires one function evaluation per iteration. The main disadvantage of the method is that the number of iterations k must be chosen at the start of the method. However, it can be proved that, given k, the length of the final interval of uncertainty is the shortest possible. Thus, in a sense, it is an optimal method.

The golden-section search method also requires one function evaluation per iteration, but the number of iterations k does not need to be chosen at the start of the method. It produces a sequence of test points

$x_1, x_2, \ldots$ and stops when a predetermined accuracy is reached.

Let $r = (\sqrt{5}-1)/2 \approx 0.618$ be the positive root of the quadratic x^2+x-1. Note that $1/r = (\sqrt{5}+1)/2 \approx 1.618$ is the famous *golden ratio*. For the Fibonacci search method, it can be proved that, for large k, the coordinates of the two initial points x_k and x_{k-1} relative to [a, b] are approximately r and $1-r$, respectively. Thus, if, at each iteration, the two points are chosen with these coordinates relative to the interval of uncertainty, the resulting method, called the golden-section search method, is an approximation to the Fibonacci search. Moreover, if a point has coordinate $1-r$ relative to [a, b], then it has coordinate r relative to $[a, a+(b-a)r]$. Similarly, if a point has coordinate r relative to [a, b], then it has coordinate $1-r$ relative to $[a+(b-a)(1-r), b]$. This property is exploited in the golden-section search method, enabling it to use one function evaluation only per iteration.

Both methods described above are guaranteed to converge whenever they are applicable and both have a linear rate of convergence (see §6.6). On the average, the length of the interval of uncertainty is multiplied by r at each iteration. For a further discussion of these methods, see an introductory numerical methods text such as (Kahaner, Moler and Nash 1989).

7.3 The Steepest Descent Method

The steepest descent (SD) method belongs to the class of gradient schemes. It is applicable whenever the partial derivatives of f exist and f has at least one local minimum. Whenever it is applicable, it is guaranteed to converge to some local minimum if the partial derivatives of f are continuous. However, it may converge slowly for multidimensional problems. Moreover, if f possesses more than one local minima, there is no guarantee to which minimum SD will converge.

At iteration k, the SD method performs a search for the minimum of f along the line $x_k - \alpha \nabla f(x_k)$, where α is a scalar variable and $-\nabla f(x_k)$ is the direction of the steepest descent of f at x_k. Note that, since α is scalar, minimizing $f(x_k - \alpha \nabla f(x_k))$ w.r.t. α is a one dimensional minimization problem. If α^* is the minimizer, x_{k+1} is taken to be $x_k - \alpha^* \nabla f(x_k)$. A brief outline of SD is given in Table 11. See (Buchanan and Turner 1992; Johnson and Riess 1982; Ortega 1988) for further details.

7.4 Conjugate Direction Methods

The definition of conjugate directions is given with respect to (w.r.t.) a symmetric positive-definite (SPD) matrix A: the vectors (directions) u and v are A-conjugate if $u^T A v = 0$. Thus, conjugate directions are orthogonal or perpendicular directions w.r.t. an inner product $(u, v) = u^T A v$ and, as such, are often associated with some shortest distance property.

The conjugate direction (CD) methods form a large class of minimization schemes. Their common characteristic is that the search direction at every iteration is conjugate to previous search directions. Proceeding along conjugate search directions guarantees, in some sense, finding the shortest path to the minimum.

The CD methods are guaranteed to converge in at most n iterations for the SPD quadratic function $f(x) = c + b^T x - \frac{1}{2} x^T A x$, where A is an $n \times n$ SPD matrix.

There are several techniques to construct conjugate directions, each one giving rise to a different CD method. The best-known is *Powell's* method (see (Buchanan and Turner 1992; Ortega 1988)).

7.5 The Conjugate Gradient Method

As the name implies, at each iteration, the conjugate gradient (CG) method takes information from the gradient of f to construct conjugate directions. Its search direction is a linear combination of the direction of steepest descent and the search direction of the previous iteration. A brief outline of CG is given in Table 12.

The CG method is guaranteed to converge in at most n iterations for the SPD quadratic function $f(x) = c + b^T x - \frac{1}{2} x^T A x$, where A is a $n \times n$ SPD matrix. See §3.2 or (Golub and Van Loan 1989; Ortega 1988)) for a more detailed discussion of this minimization technique applied to solve linear algebraic systems.

There exist several variants of the CG method, most of which are based on slightly different ways of computing the stepsize β.

7.6 Newton's Method

Newton's method for minimizing f is just Newton's method for nonlinear systems applied to solve $\nabla f(x) = 0$. The new iterate x_k is computed by $x_k = x_{k-1} - [H(x_{k-1})]^{-1} \nabla f(x_{k-1})$, where $H(x_{k-1})$ is the Hessian of f at x_{k-1}. See §6.7 for further details.

If f is convex, then the Hessian is SPD and the search direction generated by Newton's method at each iteration is a descent (downhill) direction. Thus, for any initial guess x_0, Newton's method is guaranteed to converge quadratically, provided f is sufficiently smooth.

For a general function f, there is no guarantee that Newton's method will converge for an arbitrary initial guess x_0. However, if started close enough to the minimum of a sufficiently smooth function f, Newton's method normally converges quadratically, as noted in §6.6. There exist several variants of Newton's method that improve upon the reliability of the standard scheme.

7.7 Quasi-Newton methods

At every iteration, Newton's method requires the evaluation of the Hessian and the solution of a linear system $[H(x_{k-1})]s_k = -\nabla f(x_{k-1})$ for the search direction s_k. Quasi-Newton methods update an approximation to the inverse Hessian at every iteration, thus reducing the task of solving a linear system to a simple matrix-vector multiply. The best-known of these schemes are the *Davidon-Fletcher-Powell* (DFP) and the *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) methods. See (Buchanan and Turner 1992; Dennis and Schnabel 1983) for further details.

8 Approximation

It is often desirable to find a function $f(x)$ in some class which approximates the data $\{(x_i, y_i) : i = 1, \ldots, n\}$. That is, $f(x_i) \approx y_i$, $i = 1, \ldots, n$. If f matches the data exactly, that is, f satisfies the *interpolation relations* or *interpolation conditions* $f(x_i) = y_i$, $i = 1, \ldots, n$, then f is called the *interpolating function* or the *interpolant* of the given data.

Similarly, it is often desirable to find a simple function $f(x)$ in some class which approximates a more complex function $y(x)$. We say that f *interpolates* y, or f is the *interpolant* of y, at the points x_i, $i = 1, \ldots, n$, if $f(x_i) = y(x_i)$, $i = 1, \ldots, n$. The problem of computing the interpolant f of y at the points x_i, $i = 1, \ldots, n$, reduces to the problem of computing the interpolant f of the data $\{(x_i, y_i) : i = 1, \ldots, n\}$, where $y_i = y(x_i)$.

An interpolant does not always exist and, when it does, it is not necessarily unique. However, a unique interpolant does exist in many important cases, as discussed below.

A standard approach for constructing an interpolant is to choose a set of *basis functions* $\{b_1(x), b_2(x), \ldots, b_n(x)\}$ and form a *model*

$$f(x) = \sum_{j=1}^{n} a_j b_j(x),$$

where the numbers a_j are unknown coefficients. For f to be an interpolant, it must satisfy $f(x_i) = y_i$, $i = 1, \ldots, n$, which is equivalent to

$$\sum_{j=1}^{n} a_j b_j(x_i) = y_i, \qquad i = 1, \ldots, n.$$

These n conditions form a system $B\vec{a} = \vec{y}$ of n linear equations in n unknowns, where $\vec{a} = (a_1, a_2, \ldots, a_n)^T$, $\vec{y} = (y_1, y_2, \ldots, y_n)^T$ and $B_{ij} = b_j(x_i)$, $i, j = 1, \ldots, n$. If B is non-singular, then the interpolant of the data $\{(x_i, y_i) : i = 1, \ldots, n\}$ w.r.t. the basis functions $b_1(x), b_2(x), \ldots, b_n(x)$ exists and is unique. On the other hand, if B is singular, then either the interpolant may fail to exist or there may be infinitely many interpolants.

8.1 Polynomial Approximation

Polynomial approximation is the foundation for many numerical procedures. The basic idea is that, if we want to apply some procedure to a function, such as integration (see §9), we approximate the function by a polynomial and apply the procedure to the approximating polynomial. Polynomials are often chosen as approximating functions because they are easy to evaluate (see §6.10), to integrate and to differentiate. Moreover, polynomials approximate well more complicated functions, provided the latter are sufficiently smooth. The following mathematical result ensures that arbitrarily accurate polynomial approximations exist for a broad class of functions.

Weierstrass Theorem: If $g(x) \in \mathcal{C}[a, b]$, then, for every $\epsilon > 0$, there exists a polynomial $p_n(x)$ of degree $n = n(\epsilon)$ such that $\max\{|g(x) - p_n(x)| : x \in [a, b]\} \leq \epsilon$.

8.2 Polynomial Interpolation

Techniques to construct a polynomial interpolant for a set of data $\{(x_i, y_i) : i = 1, \ldots, n\}$ are discussed below. Here we state only the following key result.

Theorem: If the points $\{x_i : i = 1, \ldots, n\}$ are distinct, then there exists a unique polynomial of degree at most $n-1$ that interpolates the data $\{(x_i, y_i) : i = 1, \ldots, n\}$. (There are no restrictions on the y_i's.)

8.2.1 Monomial Basis

One way to construct a polynomial that interpolates the data $\{(x_i, y_i) : i = 1, \ldots, n\}$ is to choose as basis functions the monomials $b_j(x) = x^{j-1}$, $j = 1, \ldots, n$, giving rise to the model $p_{n-1}(x) = a_1 + a_2 x + a_3 x^2 + \cdots + a_n x^{n-1}$. As noted above, the interpolation conditions take the form $B\vec{a} = \vec{y}$. In this case, B is the *Vandermonde matrix* for which $B_{ij} = x_i^{j-1}$, $i, j = 1, \ldots, n$, where we use the convention that $x^0 = 1$ for all x.

It can be shown that the Vandermonde matrix B is non-singular if and only if the points $\{x_i : i = 1, \ldots, n\}$ are distinct. If B is non-singular, then, of course, we can solve the system $B\vec{a} = \vec{y}$ to obtain the coefficients a_j, $j = 1, \ldots, n$, for the unique interpolant of the data.

It can also be shown that, although the Vandermonde matrix is non-singular for distinct points, it can be ill-conditioned, particularly for large n. As a result, the methods described below are often computationally much more effective than the scheme described here.

8.2.2 Lagrange Basis

An alternative to the monomial basis functions discussed above is the Lagrange basis polynomials

$$b_j(x) = l_j(x) = \prod_{\substack{i=1 \\ i \neq j}}^{n} \frac{x - x_i}{x_j - x_i} \qquad j = 1, \ldots, n,$$

which are of degree $n-1$ and satisfy

$$b_j(x_i) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

Hence $B = I$ and the system $B\vec{a} = \vec{y}$ of interpolation conditions has the obvious unique solution $a_j = y_j$, $j = 1, \ldots, n$.

Note that changing the basis from monomials to Lagrange polynomials does not change the resulting interpolating polynomial $p_{n-1}(x)$, since the interpolant is unique. It only affects the representation of $p_{n-1}(x)$.

8.2.3 Newton Basis and Divided Differences

Another useful basis is the set of Newton polynomials

$$b_j(x) = \prod_{i=1}^{j-1}(x - x_i) \qquad j = 1, \ldots, n.$$

The coefficients a_j of the interpolating polynomial written with the Newton basis are relatively easy to compute by a recursive algorithm using *divided differences*. Before describing this form of the interpolating polynomial, though, we must introduce divided differences.

Given a function f with $f(x_i) = y_i$, $i = 1, \ldots, n$, define the divided difference with one point by

$$f[x_i] = y_i \qquad i = 1, \ldots, n.$$

If $x_{i+1} \neq x_i$, define the divided difference with two points by

$$f[x_i, x_{i+1}] = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \qquad i = 1, \ldots, n-1.$$

If $x_i \neq x_{i+k}$, define the divided difference with $k+1$ points by

$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f[x_{i+1}, x_{i+2}, \ldots, x_{i+k}] - f[x_i, x_{i+1}, \ldots, x_{i+k-1}]}{x_{i+k} - x_i} \qquad i = 1, \ldots, n-k.$$

We can extend this definition of divided differences to sets $\{x_i : i = 1, \ldots, n\}$ with repeated values by noting that

$$\lim_{x_{i+1} \to x_i} f[x_{i+1}, x_i] = \lim_{x_{i+1} \to x_i} \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = f'(x_i)$$

So, if $x_i = x_{i+1}$, we define $f[x_i, x_{i+1}] = f'(x_i)$. Similarly, it can be shown that

$$\lim_{\substack{x_{i+1} \to x_i \\ \vdots \\ x_{i+k} \to x_i}} f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f^{(k)}(x_i)}{k!}$$

So, if $x_i = x_{i+1} = \cdots = x_{i+k}$, we define $f[x_i, x_{i+1}, \ldots, x_{i+k}] = f^{(k)}(x_i)/k!$.

Using divided differences, the coefficients a_j can be computed as $a_j = f[x_1, \ldots, x_j]$. An advantage of the divided difference form is that it extends easily to the interpolation of derivatives as well as function values, as discussed below. Another advantage is that, if the coefficients a_j, $j = 1, \ldots, n$, are computed from n data points, it is easy to add more data points and construct a higher degree interpolating polynomial without redoing the whole computation, since the new data can be added easily to the existing divided difference table and the interpolating polynomial extended.

8.3 Polynomial Interpolation with Derivative Data

In this section, we consider *Hermite* or *osculatory interpolation* which requires that a function and its first derivative be interpolated at the points $\{x_i : i = 1, \ldots, n\}$. The key result is stated below.

Theorem: Given the data $\{(x_i, y_i, y_i') : i = 1, \ldots, n\}$, where the points $\{x_i : i = 1, \ldots, n\}$ are distinct, there exists a unique polynomial interpolant $p_{2n-1}(x)$ of degree at most $2n-1$ that satisfies $p_{2n-1}(x_i) = y_i$, $i = 1, \ldots, n$, and $p'_{2n-1}(x_i) = y_i'$, $i = 1, \ldots, n$.

The techniques used to construct such a polynomial are similar to those described in §8.2. The following choices of basis functions are often used.

- Monomials: $b_j(x) = x^{j-1}$, $j = 1, \ldots, 2n-1$.
- Generalized Lagrange basis polynomials: $b_j(x) = [1-2(x-x_j)l_j'(x_j)][l_j(x)]^2$, $j = 1, \ldots, n$ and $b_{n+j}(x) = (x-x_j)[l_j(x)]^2$, $j = 1, \ldots, n$,
- Newton basis polynomials:

$$b_j(x) = \prod_{i=1}^{j-1} (x - x_i)^2, \qquad j = 1, \ldots, n$$

and

$$b_{n+j}(x) = \left(\prod_{i=1}^{j-1} (x - x_i)^2 \right) (x - x_j), \qquad j = 1, \ldots, n.$$

More general forms of polynomial interpolants are discussed in some numerical methods books. See for example (Davis 1975).

8.4 The Error in Polynomial Interpolation

Two key results for the error in polynomial interpolation are given in the following two theorems. For their proofs, see an introductory numerical methods text such as (Dahlquist and Björck 1974; Johnson and Riess 1982; Stoer and Bulirsch 1980). Before stating the results, though, we must define $\text{spr}[x, x_1, x_2, \ldots, x_n]$ to be the smallest interval containing $x, x_1, x_2, \ldots, x_n$.

Theorem: Let $g(x) \in \mathcal{C}^n$ and $p_{n-1}(x)$ be the polynomial of degree at most $n-1$ that interpolates $g(x)$ at the n distinct points $x_1, x_2, \ldots, x_n$. Then, for any x,

$$g(x) - p_{n-1}(x) = \frac{g^{(n)}(\xi)}{n!} \prod_{i=1}^{n} (x - x_i)$$

where $g^{(n)}(x)$ is the n^{th} derivative of $g(x)$ and ξ is some point in $\text{spr}[x, x_1, x_2, \ldots, x_n]$.

Theorem: Let $g(x) \in \mathcal{C}^{2n}$ and $p_{2n-1}(x)$ be the polynomial of degree at most $2n-1$ that interpolates $g(x)$ and $g'(x)$ at the n distinct points $x_1, x_2, \ldots, x_n$. Then, for any x,

$$g(x) - p_{2n-1}(x) = \frac{g^{(2n)}(\xi)}{(2n)!} \prod_{i=1}^{n} (x - x_i)^2$$

where $g^{(2n)}(x)$ is the $2n^{\text{th}}$ derivative of $g(x)$ and ξ is some point in $\text{spr}[x, x_1, x_2, \ldots, x_n]$.

Note that there is a close relationship between the error in polynomial interpolation and the error in a Taylor series. As a result, polynomial interpolation is normally effective if and only if g can be approximated well by a Taylor series.

More specifically, the polynomial interpolation error can be large if the derivative appearing in the error formula is big or if $\text{spr}[x, x_1, x_2, \ldots, x_n]$ is big, particularly if the point x of evaluation is close to an end-point of $\text{spr}[x_1, x_2, \ldots, x_n]$ or outside this interval.

8.5 Piecewise Polynomials and Splines

Given a set of *knots* or *grid points* $\{x_i : i = 1, \ldots, n\}$ satisfying $a = x_0 < x_1 < \cdots < x_n = b$, $s(x)$ is a *piecewise polynomial* (PP) of degree N w.r.t. the knots $\{x_i, i = 0, \ldots, n\}$ if $s(x)$ is a polynomial of degree N on each interval (x_{i-1}, x_i), $i = 1, \ldots, n$. A polynomial of degree N is always a PP of degree N, but the converse is not necessarily true.

A *spline* $s(x)$ of degree N is a PP of degree N. The term spline usually implies the continuity of $s(x), s'(x), \ldots, s^{(N-1)}(x)$ at the knots $\{x_0, x_1, \ldots, x_n\}$. In this case, $s \in \mathcal{C}^{N-1}$, the space of continuous functions with $N-1$ continuous derivatives. Sometimes, though, the terms PP and spline are used interchangeably.

Let $s(x)$ be a PP of degree N and assume $s(x), s'(x), \ldots, s^{(K)}(x)$ are continuous at the knots $\{x_0, x_1, \ldots, x_n\}$. Since $s(x)$ is a polynomial of degree N on each of the n subintervals, (x_i, x_{i-1}), $i = 1, \ldots, n$, it is defined by $D = n(N+1)$ coefficients. To determine these coefficients, we take into account the continuity conditions that s and its K derivatives must satisfy at the $n-1$ interior knots $\{x_1, \ldots, x_{n-1}\}$. There are $K+1$ such conditions at each interior knot, giving rise to $C = (n-1)(K+1)$ conditions. Thus, we need $M = D - C = n(N-K) + K + 1$ properly-chosen additional conditions to determine the coefficients of s.

In the following we give examples of PPs and splines and their associated basis functions.

8.5.1 Constant Splines

The constant PP model function

$$\phi(x) = \begin{cases} 1 & \text{for } 0 \le x \le 1 \\ 0 & \text{elsewhere} \end{cases}$$

is a constant spline w.r.t. the knots $\{0, 1\}$. Note that it is not continuous at the knots, so $\phi \in \mathcal{C}^{-1}$.

The constant PP functions

$$\phi_i(x) = \phi((x-a)/h - i + 1), \quad i = 1, \ldots, n$$

are constant splines w.r.t. the evenly-spaced knots $\{x_i = a + ih : i = 0, \ldots, n\}$, where $h = (b-a)/n$, and form a set of basis functions for the space of constant splines w.r.t. these knots.

8.5.2 Linear Splines

The linear PP model function

$$\phi(x) = \begin{cases} x & \text{for } 0 \le x \le 1 \\ 2 - x & \text{for } 1 \le x \le 2 \\ 0 & \text{elsewhere} \end{cases}$$

is a linear spline w.r.t. the knots $\{0, 1, 2\}$. Note that ϕ is continuous, but ϕ' does not exist at the knots, so $\phi \in \mathcal{C}^0$.

The linear PP functions

$$\phi_i(x) = \phi((x - a)/h - i + 1), \quad i = 0, \ldots, n \tag{39}$$

are linear splines w.r.t. the evenly-spaced knots $\{x_i = a + ih : i = 0, \ldots, n\}$, where $h = (b - a)/n$, and form a set of basis functions for the space of linear splines w.r.t. these knots.

8.5.3 Quadratic Splines

The quadratic PP model function

$$\phi(x) = \begin{cases} x^2/2 & \text{for } 0 \le x \le 1 \\ (x^2 - 3(x-1)^2)/2 & \text{for } 1 \le x \le 2 \\ (x^2 - 3(x-1)^2 + 3(x-2)^2)/2 & \text{for } 2 \le x \le 3 \\ 0 & \text{elsewhere} \end{cases}$$

is a quadratic spline w.r.t. the knots $\{0, 1, 2, 3\}$. Note that ϕ and ϕ' are continuous, but ϕ'' does not exist at the knots, so $\phi \in \mathcal{C}^1$.

The quadratic PP functions

$$\phi_i(x) = \phi((x - a)/h - i + 2), \quad i = 0, \ldots, n + 1 \tag{40}$$

are quadratic splines w.r.t. the evenly-spaced knots $\{x_i = a + ih : i = 0, \ldots, n\}$, where $h = (b - a)/n$, and form a set of basis functions for the space of quadratic splines w.r.t. these knots.

8.5.4 Quadratic Piecewise Polynomials

The quadratic PP model functions

$$\phi(x) = \begin{cases} x(2x - 1) & \text{for } 0 \le x \le 1 \\ (x - 2)(2x - 3) & \text{for } 1 \le x \le 2 \\ 0 & \text{elsewhere} \end{cases}$$

$$\psi(x) = \begin{cases} -4x(x - 1) & \text{for } 0 \le x \le 1 \\ 0 & \text{elsewhere} \end{cases}$$

are continuous, but neither ϕ' nor ψ' exist at their knots $\{0,\ 1,\ 2\}$ and $\{0,\ 1\}$, respectively. So ϕ and $\psi \in \mathcal{C}^0$.

The functions

$$\phi_i(x) = \left\{ \begin{array}{ll} \phi((x-a)/h - i/2 + 1) & \text{for } i \text{ even} \\ \psi((x-a)/h - (i+1)/2 + 1) & \text{for } i \text{ odd} \end{array} \right\} \qquad i = 0, \ldots, 2n$$

are quadratic PPs w.r.t. the evenly-spaced knots $\{x_i = a + ih : i = 0, \ldots, n\}$, where $h = (b-a)/n$. Note ϕ_i is continuous, but ϕ_i' does not exist at the knots, so $\phi_i \in \mathcal{C}^0$. These functions form a basis for the space of quadratic PPs in $\mathcal{C}^0$ w.r.t. these knots.

8.5.5 Cubic Splines

The cubic PP model function

$$\phi(x) = \left\{ \begin{array}{ll} x^3/6 & \text{for } 0 \le x \le 1 \\ (x^3 - 4(x-1)^3)/6 & \text{for } 1 \le x \le 2 \\ (x^3 - 4(x-1)^3 + 6(x-2)^3)/6 & \text{for } 2 \le x \le 3 \\ (x^3 - 4(x-1)^3 + 6(x-2)^3 - 4(x-3)^3)/6 & \text{for } 3 \le x \le 4 \\ 0 & \text{elsewhere} \end{array} \right.$$

is a cubic spline w.r.t. the knots $\{0,\ 1,\ 2,\ 3,\ 4\}$. Note that ϕ, ϕ' and ϕ'' are continuous, but ϕ''' does not exist at the knots, so $\phi \in \mathcal{C}^2$.

The cubic PP functions

$$\phi_i(x) = \phi((x-a)/h - i + 2), \quad i = -1, \ldots, n+1 \tag{41}$$

are cubic splines w.r.t. the evenly-spaced knots $\{x_i = a + ih : i = 0, \ldots, n\}$, where $h = (b-a)/n$, and form a set of basis functions for the space of cubic splines w.r.t. these knots.

8.5.6 Cubic Hermite Piecewise Polynomials

The cubic PP model functions

$$\phi(x) = \left\{ \begin{array}{ll} x^2(3-2x) & \text{for } 0 \le x \le 1 \\ (2-x)^2(2x-1) & \text{for } 1 \le x \le 2 \\ 0 & \text{elsewhere} \end{array} \right.$$

$$\psi(x) = \left\{ \begin{array}{ll} x^2(x-1) & \text{for } 0 \le x \le 1 \\ (2-x)^2(x-1) & \text{for } 1 \le x \le 2 \\ 0 & \text{elsewhere} \end{array} \right.$$

are in $\mathcal{C}^1$ since ϕ, ϕ', ψ and ψ' are all continuous at the knots $\{0,\ 1,\ 2\}$, but neither ϕ'' nor ψ'' exist at the knots.

The functions

$$\phi_i(x) = \left\{ \begin{array}{ll} \phi((x-a)/h - i/2 + 1) & \text{for } i \text{ even} \\ \psi((x-a)/h - (i-1)/2 + 1) & \text{for } i \text{ odd} \end{array} \right\} \qquad i = 0, \ldots, 2n+1 \tag{42}$$

are cubic PPs w.r.t. the evenly-spaced knots $\{x_i = a + ih : i = 0, \ldots, n\}$, where $h = (b-a)/n$. Note ϕ_i and ϕ' are continuous, but ϕ_i'' does not exist at the knots, so $\phi_i \in \mathcal{C}^1$. These functions form a basis for the space of cubic PPs in $\mathcal{C}^1$ w.r.t. these knots.

8.6 Piecewise Polynomial Interpolation

Piecewise polynomials, including splines, are often used for interpolation, especially for large data sets. The main advantage in using PPs, instead of polynomials, to interpolate a function $g(x)$ at n points, where $n \gg 1$, is that the error of a PP interpolant does not depend on the n^{th} derivative of $g(x)$, but rather on a low-order derivative of $g(x)$. Usually, if $g(x) \in \mathcal{C}^{N+1}[a,b]$ and $s_N(x)$ is a PP of degree N in $\mathcal{C}^K$ that interpolates $g(x)$ at $n(N-K)+K+1$ properly chosen points, then the interpolation error at any point $x \in [a,b]$ satisfies

$$|g(x) - s_N(x)| \leq C\, h^{N+1} \max_{a \leq \xi \leq b} |g^{(N+1)}(\xi)|$$

for some constant C independent of $h = \max\{x_i - x_{i-1} : i = 1, \ldots, n\}$.

Another advantage of PP interpolation is that it leads either to simple relations that define the PP interpolant or gives rise to banded systems which can be solved easily for the coefficients of the PP interpolant by the techniques described in §2.7.

In the following subsections, we give examples of PP interpolation. We assume throughout that $h = (b-a)/n$ and $x_i = a + i\,h$ for $i = 0, \ldots, n$, but these restrictions can be removed easily.

An introduction to PP interpolation is presented in many introductory numerical analysis textbooks, such as (Johnson and Riess 1982). For a more detailed discussion, see an advanced text such as (Prenter 1975) or (de Boor 1978).

8.6.1 Linear Spline Interpolation

Let $\phi_i(x)$ be defined by (39). The function

$$s_1(x) = \sum_{i=0}^{n} c_i \phi_i(x)$$

with $c_i = g(x_i)$ is the unique linear spline that interpolates $g(x)$ at the knots $\{x_i : i = 0, \ldots, n\}$.

8.6.2 Quadratic Spline Interpolation

Let $\phi_i(x)$ be defined by (40). The function

$$s_2(x) = \sum_{i=0}^{n+1} c_i \phi_i(x)$$

is the unique quadratic spline that interpolates $g(x)$ at the $n+2$ points x_0, $(x_{i-1}+x_i)/2$, $i = 1, \ldots, n$, and x_n, if the coefficients c_i, $i = 0, \ldots, n+1$, are chosen so that they satisfy the $n+2$ conditions $s_2(x_0) = g(x_0)$, $s_2((x_{i-1}+x_i)/2) = g((x_{i-1}+x_i)/2)$, $i = 1, \ldots, n$, and $s_2(x_n) = g(x_n)$. These conditions give rise to a tridiagonal linear system of $n+2$ equations in the $n+2$ unknowns c_i, $i = 0, \ldots, n+1$:

$$\begin{pmatrix} 0.5 & 0.5 & & & & & \\ 0.125 & 0.75 & 0.125 & & & & \\ & 0.125 & 0.75 & 0.125 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 0.125 & 0.75 & 0.125 & \\ & & & & 0.5 & 0.5 & \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \\ c_{n+1} \end{pmatrix} = \begin{pmatrix} g(x_0) \\ g((x_0+x_1)/2) \\ g((x_1+x_2)/2) \\ \vdots \\ g((x_{n-1}+x_n)/2) \\ g(x_n) \end{pmatrix}$$

Since this system is diagonally dominant in all rows and strictly diagonally dominant in all rows expect the first and last, it is nonsingular and has a unique solution. Moreover, if the first and last equations are multiplied by .25, the resulting system is symmetric positive-definite. So the coefficients $\{c_i\}$ of the associated quadratic spline interpolant can be computed easily by the techniques described in §2.7.

8.6.3 Cubic Spline Interpolation

Let $\phi_i(x)$ be defined by (41). The function

$$s_{32}(x) = \sum_{i=-1}^{n+1} c_i \phi_i(x)$$

is a cubic spline that interpolates $g(x)$ at the $n+1$ points x_i, $i = 0, \ldots, n$, if the coefficients c_i, $i = -1, \ldots, n+1$, are chosen so that they satisfy the $n+1$ conditions $s_{32}(x_i) = g(x_i)$, $i = 0, \ldots, n$. These conditions give rise to a linear system of $n+1$ equations in the $n+3$ unknowns c_i, $i = -1, \ldots, n+1$. To determine the unknown coefficients uniquely, two more appropriately chosen conditions (equations) are needed. The standard choices are:

(a) $s''_{32}(x_0) = 0$ and $s''_{32}(x_n) = 0$, giving rise to the *natural* or *minimum curvature* cubic spline interpolant,

(b) $s'_{32}(x_0) = g'(x_0)$ and $s'_{32}(x_n) = g'(x_n)$, giving rise to a more accurate cubic spline interpolant, but one which requires knowledge of $g'(x)$ (or a good approximation to it) at both endpoints,

(c) the *not-a-knot* conditions which force $s_{32}(x)$ to have a continuous third derivative at the knots x_1 and x_{n-1}.

In all three cases, the resulting linear system of $n+3$ equations in $n+3$ unknowns is almost tridiagonal. More specifically, it is tridiagonal with the exception of the two rows corresponding to the extra conditions. So the coefficients $\{c_i\}$ of the associated cubic spline interpolant can be computed easily by the techniques described in §2.7.

8.6.4 Cubic Hermite Piecewise Polynomial Interpolation

Let $\phi_i(x)$ be defined by (42). The function

$$s_{31}(x) = \sum_{i=0}^{2n+1} c_i \phi_i(x)$$

with $c_{2i} = g(x_i)$, $i = 0, \ldots, n$, and $c_{2i+1} = g'(x_i)$, $i = 0, \ldots, n$, is the unique Hermite PP that interpolates $g(x)$ and its derivative at the $n+1$ points x_i, $i = 0, \ldots, n$.

8.7 Least Squares Approximation

It is possible to construct a function $f(x)$ which approximates the data $\{(x_i, y_i) : i = 0, \ldots, m\}$ in the sense that $f(x_i) - y_i$ is "small" for $i = 0, \ldots, m$, but $f(x_i) \neq y_i$ in general. One way is to construct an f in some class of functions that minimizes

$$\sum_{i=0}^{m} w_i (f(x_i) - y_i)^2$$

for some positive *weights* w_i, $i = 0, \ldots, m$. Such an f is called a *discrete least squares* approximation to the data.

Similarly, it is possible to construct a function $f(x)$ which approximates a continuous function $y(x)$ on an interval $[a, b]$ in the sense that $|f(x) - y(x)|$ is "small" for all $x \in [a, b]$. One way is to construct an f in some class of functions that minimizes

$$\int_a^b w(x)(f(x) - y(x))^2 \, dx$$

for some positive and continuous *weight function* $w(x)$ on (a, b). Such an f is called a *continuous least squares* approximation to y.

Often, f is chosen to be a polynomial of degree $n < m$. (For the discrete least squares problem, if f is a polynomial of degree $n = m$, then f interpolates the data.)

Let f and $g \in \mathcal{C}[a, b]$. We denote the *discrete inner product* of f and g at distinct points x_i, $i = 0, \ldots, m$, w.r.t. the weights w_i, $i = 0, \ldots, m$, by

$$(f, g) = \sum_{i=0}^{m} w_i f(x_i) g(x_i)$$

and the *continuous inner product* of f and g on $[a,b]$ w.r.t. weight function $w(x)$ by

$$(f,g) = \int_a^b w(x)f(x)g(x)\,dx$$

Given an inner product $(\cdot,\cdot)$, as above, we denote the *norm* of f by $\|f\| = \sqrt{(f,f)}$. Thus we have the *discrete norm*

$$\|f\| = \sqrt{\sum_{i=0}^{m} w_i f(x_i)^2}$$

and the *continuous norm*

$$\|f\| = \sqrt{\int_a^b w(x)f(x)^2\,dx}$$

Therefore, the problem of constructing a least squares approximation $f(x)$ to a given set of data $\{(x_i,y_i)\}$ or to a given function $y(x)$ is to construct an $f(x)$ that minimizes the norm, discrete or continuous, respectively, of the error $\|f-y\|$.

We note that the "discrete inner product" is not strictly speaking an inner product in all cases, since it may fail to satisfy the property that $(f,f)=0$ implies $f(x)=0$ for all x. However, if we restrict the class of functions to which f belongs appropriately, then $(\cdot,\cdot)$ is a true inner product. For example, if we restrict f to the class of polynomials of degree at most n and if $n<m$, then $(f,f)=0$ implies $f(x)=0$ for all x. Similar remarks apply to the discrete norm.

Before giving the main theorem on how to construct the least squares polynomial approximation to a given set of data or to a given function, we introduce orthogonal polynomials and the Gram-Schmidt process to construct them.

8.7.1 Orthogonal Polynomials

A set of $n+1$ polynomials $\{q_i(x) : i=0,\ldots,n\}$ is *orthogonal* w.r.t. the inner product $(\cdot,\cdot)$ if $(q_i,q_j)=0$ for $i\neq j$. A set of $n+1$ orthogonal polynomials $\{q_i(x) : i=0,\ldots,n\}$ is *orthonormal* w.r.t. the inner product $(\cdot,\cdot)$ if in addition $(q_i,q_i)=1$ for $i=0,\ldots,n$.

8.7.2 The Gram-Schmidt Orthogonalization Algorithm

The Gram-Schmidt algorithm applied to a set of $n+1$ linearly independent polynomials $\{p_j : j=0,\ldots,n\}$ generates a set of $n+1$ orthonormal polynomials $\{q_i(x) : i=0,\ldots,n\}$ and a set of $n+1$ orthogonal polynomials $\{s_i(x) : i=0,\ldots,n\}$. Often, the set of $n+1$ linearly independent polynomials $\{p_j : j=0,\ldots,n\}$ is chosen to be the set of monomials $\{x^j : j=0,\ldots,n\}$.

The Gram-Schmidt algorithm for polynomials is similar to the Gram-Schmidt algorithm for matrices described in §4.6. The reader may refer to an introductory numerical methods text such as (Johnson and Riess 1982) for more details. Here we note only that the role of inner product of vectors in the algorithm in §4.6 is replaced by the inner product, discrete or continuous, of functions as defined in §8.7.1.

8.7.3 Constructing the Least Squares Polynomial Approximation

The following result is proved in many introductory numerical methods books. See for example (Johnson and Riess 1982).

Theorem: Assume that we are given either a continuous function $y(x)$ on $[a,b]$ or a data set $\{(x_i, y_i) : i = 0, \ldots, m\}$. Let $\{q_j : j = 0, \ldots, n\}$ be a set of orthonormal polynomials w.r.t. an inner product $(\cdot, \cdot)$ appropriate for the given data and assume $\{q_j : j = 0, \ldots, n\}$ spans $\{x^i : i = 0, \ldots, n\}$, where $n < m$ for the discrete problem. Then

$$p^*(x) = \sum_{j=0}^{n} (y, q_j)\, q_j(x)$$

is the least squares polynomial of degree at most n. It is optimal in the sense that, if $p(x)$ is any other polynomial of degree at most n, then $\|y - p^*\| < \|y - p\|$, where $\|\cdot\|$ is the norm associated with the inner product $(\cdot, \cdot)$.

As noted in §8.7.2, a set of orthonormal polynomials $\{q_j : j = 0, \ldots, n\}$ that spans $\{x^i : i = 0, \ldots, n\}$ can be constructed by the Gram-Schmidt algorithm applied to the monomial basis polynomials $\{x^i : i = 0, \ldots, n\}$.

9 Numerical Integration — Quadrature

In this section, we consider formulas for approximating integrals of the form

$$I(f) = \int_a^b f(x)\, dx.$$

Such formulas are often called quadrature rules. We assume a and b are finite and f is smooth in most cases, but we briefly discuss infinite integrals and singularities in §9.5.

In many practical problems, $f(x)$ is given either as a set of values $f(x_1), \ldots, f(x_n)$ or $f(x)$ is hard or impossible to integrate exactly. In these cases, the integral may be approximated by numerical techniques, which often take the form

$$Q(f) = \sum_{i=1}^{n} w_i f(x_i).$$

Such a formula is called a *quadrature rule*, the $\{w_i\}$ are called *weights* and the $\{x_i\}$ are called *abscissae* or *nodes*.

Most quadrature rules are derived by first approximating f by a simpler function, frequently a polynomial, and then integrating the simpler function. Thus, the area under the curve f, which is the exact value of $I(f)$, is approximated by the area under the curve of the simpler function.

For a more detailed discussion of the topics in this section, see an introductory numerical methods text such as (Conte and de Boor 1980; Dahlquist and Björck 1974; Johnson and Riess 1982; Stoer and Bulirsch 1980).

9.1 Simple Quadrature Rules

Several simple quadrature rules are listed below.

- The *rectangle rule* approximates $I(f)$ by the area under the constant $y = f(a)$ or $y = f(b)$.
- The *midpoint rule* approximates $I(f)$ by the area under the constant $y = f((a+b)/2)$.
- The *trapezoidal rule* approximates $I(f)$ by the area under the straight line joining the points $(a, f(a))$ and $(b, f(b))$.
- *Simpson's rule* approximates $I(f)$ by the area under the quadratic that interpolates $(a, f(a))$, $(m, f(m))$ and $(b, f(b))$, where $m = (a+b)/2$.
- The *corrected trapezoidal rule* approximates $I(f)$ by the area under the cubic Hermite that interpolates $(a, f(a), f'(a))$ and $(b, f(b), f'(b))$.
- Newton-Cotes rules are discussed in §9.1.1 below.
- Gaussian rules are discussed in §9.1.2 below.

The formula $Q(f)$ and the associated error $I(f) - Q(f)$ for each quadrature rule listed above are given in Table 13, where n is the number of function and derivative evaluations, d is the polynomial degree of the quadrature rule (see §9.1.1 below), η is an unknown point in $[a, b]$ (generally different for each rule), $m = (a+b)/2$ is the midpoint of the interval $[a, b]$, C and K are some constants. For the derivation of these quadrature rules and their associated error formulas, see an introductory numerical methods text such as (Conte and de Boor 1980; Dahlquist and Björck 1974; Johnson and Riess 1982; Stoer and Bulirsch 1980).

9.1.1 Some Definitions

A quadrature rule which is based on integrating a polynomial interpolant is called an *interpolatory rule*. All the simple quadrature rules listed in Table 13 are interpolatory. Writing the polynomial interpolant in Lagrange form and integrating it, we see immediately that the weights w_i do not depend on the function f, but only on the abscissae $\{x_i : i = 1, \ldots, n\}$.

Quadrature rules are, in general, not exact. An error formula for an interpolatory rule can often be derived by integrating the associated polynomial interpolant error. Error formulas for some simple quadrature rules are listed in Table 13.

A quadrature rule which is exact for all polynomials of degree d or less, but is not exact for all polynomials of degree $d+1$, is said to have *polynomial degree* d. An interpolatory rule based on n function and derivative values has polynomial degree at least $n-1$.

Quadrature rules that include the end-points of the interval of integration $[a, b]$ as abscissae are called *closed rules*, while those that do not include the end-points are called *open rules*. An advantage of open rules is that they can be applied to integrals with singularities at the end-points, whereas closed rules usually can not.

Interpolatory rules based on equidistant abscissae are called *Newton-Cotes* rules. This class includes the rectangle, midpoint, trapezoidal, corrected trapezoidal and Simpson's rules. Both open and closed Newton-Cotes quadrature rules exist.

9.1.2 Gaussian Quadrature Rules

A quadrature rule

$$Q(f) = \sum_{i=1}^{n} w_i f(x_i)$$

is fully determined by n, the abscissae $\{x_i : i = 1, \ldots, n\}$ and the weights $\{w_i : i = 1, \ldots, n\}$. Gauss showed that, given n and the end-points a and b of the integral,

1. there exists a unique set of abscissae $\{x_i\}$ and weights $\{w_i\}$ which give a quadrature rule — called the Gaussian quadrature rule — that is exact for all polynomials of degree $2n-1$ or less,
2. no quadrature rule with n abscissae and n weights is exact for all polynomials of degree $2n$,
3. the weights $\{w_i\}$ of the Gaussian quadrature rule are all positive,
4. the Gaussian quadrature rule is open,
5. the abscissae $\{x_i\}$ of the Gaussian quadrature rule are the roots of the shifted Legendre polynomial $q_n(x)$ of degree n, which is the unique monic polynomial of degree n that is orthogonal to all polynomials of degree $n-1$ or less w.r.t. the continuous inner product

$$(f, g) = \int_a^b f(x)g(x)\,dx$$

(see §8.7.1),

6. the Gaussian quadrature rule is interpolatory, i.e., it can be derived by integrating the polynomial of degree $n-1$ that interpolates f at the abscissae $\{x_i\}$.

Thus, Gauss derived the class of open interpolatory quadrature rules of maximum polynomial degree $d = 2n-1$. As noted above, these formulas are called *Gaussian quadrature rules* or *Gauss-Legendre quadrature rules*.

9.1.3 Translating the Interval of Integration

The weights and abscissae of simple quadrature rules are usually given w.r.t. a specific interval of integration, such as $[0, 1]$ or $[-1, 1]$. However, the weights and abscissae can be transformed easily to obtain a related quadrature rule appropriate for some other interval.

One simple way to do this is based on the linear change of variables $\hat{x} = \beta(x-a)/(b-a) + \alpha(b-x)/(b-a)$ which leads to the relation

$$\int_\alpha^\beta f(\hat{x})\,d\hat{x} = \int_a^b f\left(\beta\frac{x-a}{b-a} + \alpha\frac{b-x}{b-a}\right)\frac{\beta-\alpha}{b-a}\,dx \tag{43}$$

So, if we are given a quadrature rule on $[a, b]$ with weights $\{w_i : i = 1, \ldots, n\}$ and abscissae $\{x_i : i = 1, \ldots, n\}$, but we want to compute

$$\int_{\alpha}^{\beta} f(\hat{x})\, d\hat{x},$$

we can apply the quadrature rule to the integral on the right side of (43). An equivalent way of viewing this is that we have developed a related quadrature rule for the interval of integration $[\alpha, \beta]$ with weights and abscissae

$$\hat{w}_i = \frac{\beta - \alpha}{b - a}\, w_i \qquad \text{and} \qquad \hat{x}_i = \beta \frac{x_i - a}{b - a} + \alpha \frac{b - x_i}{b - a}$$

for $i = 1, \ldots, n$. Note that, because we have used a linear change of variables, the original rule for $[a, b]$ and the related one for $[\alpha, \beta]$ have the same polynomial degree.

9.1.4 Comparison of Gaussian and Newton-Cotes Quadrature Rules

We list some similarities and differences between Gaussian and Newton-Cotes quadrature rules below.

1. The weights of a Gaussian rule are all positive, which contributes to the stability of the formula. High order Newton-Cotes rules typically have both positive and negative weights, which is less desirable, since it leads to poorer stability properties.

2. Gaussian rules are open, whereas there are both open Newton-Cotes rules and closed Newton-Cotes rules.

3. Gaussian rules attain the maximum possible polynomial degree $2n - 1$ for a formula with n weights and abscissae, whereas the polynomial degree d of a Newton-Cotes rule satisfies $n - 1 \leq d \leq 2n - 1$ and the upper bound can be obtained for $n = 1$ only.

4. The abscissae and weights of Gaussian rules are often irrational numbers and hard to remember, but they are not difficult to compute. The abscissae of a Newton-Cotes rule are easy to remember and the weights are simple to compute.

5. The set of abscissae for a n-point Gaussian rule and for a m-point Gaussian rule are almost disjoint for all $n \neq m$. Thus we can not reuse function evaluations performed for one Gaussian rule in another Gaussian rule. Appropriately chosen pairs of Newton-Cotes rules can share function evaluations effectively.

6. Both Gaussian and Newton-Cotes rules are interpolatory.

9.2 Composite (Compound) Quadrature Rules

To increase the accuracy of a numerical approximation to an integral, we could use a rule with more weights and abscissae. This often works with Gaussian rules, if f is sufficiently smooth, but it is not advisable with Newton-Cotes rules, for example, because of stability problems associated with high-order formulas in this class.

Another effective way to achieve high accuracy is to use *composite* quadrature rules, often also called *compound* quadrature rules. In these schemes, the interval of integration $[a, b]$ is subdivided into *panels* (or subintervals) and on each panel the same simple quadrature rule is applied. If the associated simple quadrature rule is interpolatory, then this approach leads to the integration of a piecewise polynomial (PP) interpolant. Thus, using a composite quadrature rule, instead of a simple one with high polynomial degree, leads to many of the same benefits that are obtained in using PP interpolants compared to high-degree polynomial interpolants (see §8.5).

Table 14 summarizes the composite quadrature rules and the associated error formulas. In the table, n is the number of function and derivative evaluations, d is the polynomial degree of the quadrature rule, η is an unknown point in $[a, b]$ (in general different for each rule), $h = (b - a)/s$ is the stepsize of each panel, s is the number of panels and PP stands for piecewise polynomial. Composite quadrature rules based on Gaussian or Newton-Cotes rules can also be used, although they are not listed in Table 14.

9.3 Adaptive Quadrature

We see from Table 14 of composite quadrature rules that the smaller the stepsize h of a panel the smaller the expected error. It is often the case that an approximation to the integral

$$I(f) = \int_a^b f(x)\, dx$$

is needed to within a specified accuracy ϵ. In this case, *adaptive quadrature* is often used. Such schemes refine the grid (or collection of panels) until an estimate of the total error in the integration is within the desired precision ϵ. Adaptive quadrature is particularly useful when the behaviour of the function f varies significantly in the interval of integration $[a, b]$, since the scheme can use a fine grid where f is hard to integrate and a coarse grid where it is easy, leading to an efficient and accurate quadrature procedure.

Table 15 gives a simple general recursive procedure for adaptive quadrature. We assume that we can make use of a routine LQM (Local Quadrature Module) which implements a quadrature rule in some interval $[a, b]$ and returns Q, an approximation to the integral, and E, an estimate of the error. In the next section, we discuss how an error estimate may be obtained.

We note that the adaptive quadrature procedure shown in Table 15 does not illustrate how to reuse function evaluations where possible. An effective adaptive quadrature routine should do this, since function evaluations are often the most computationally expensive part of the procedure.

9.4 Romberg Integration and Error Estimation

As discussed in the last subsection, adaptive quadrature requires an error estimate. As an illustration, we consider how one may be obtained for the composite trapezoidal rule.

Let $T_s(f)$ denote the composite trapezoidal rule approximation to

$$I(f) = \int_a^b f(x)\,dx$$

using s panels and let $E_s = I(f) - T_s(f)$ be the associated error. Based on the error formula in Table 13, we expect E_{2s} to be about 4 times smaller than E_s, assuming that $f''(x)$ does not vary too much. That is,

$$\begin{aligned} I(f) - T_s(f) &= E_s \\ I(f) - T_{2s}(f) &= E_{2s} \approx \frac{1}{4}E_s. \end{aligned}$$

Subtracting these two equations, we get

$$T_{2s}(f) - T_s(f) = E_s - E_{2s} \approx \frac{3}{4}E_s.$$

So $E_s \approx 4(T_{2s}(f) - T_s(f))/3$ and $E_{2s} \approx (T_{2s}(f) - T_s(f))/3$. Thus, by applying the composite trapezoidal rule first with s and then with $2s$ panels, we obtain estimates of the error in both $T_s(f)$ and $T_{2s}(f)$.

If we add the estimate of the error $E_s = I(f) - T_s(f)$ to $T_s(f)$ we often obtain a better approximation to $I(f)$ than either $T_s(f)$ or $T_{2s}(f)$. That is, $\hat{T}_s(f) = T_s(f) + 4(T_{2s}(f) - T_s(f))/3 = (4T_{2s}(f) - T_s(f))/3$ is often a better approximation to $I(f)$ than either $T_s(f)$ or $T_{2s}(f)$, particularly if the function f is smooth and the grid is fine. Thus, by applying the compound trapezoidal rule with s and $2s$ panels and taking an appropriate linear combination of the two approximations, we construct a better approximation to $I(f)$. To be more specific, it can be shown that this eliminates the lowest order term in the error E_s or E_{2s}. This process can be repeated to eliminate the next higher-order term in the error and so on. In addition, it can be generalized easily to other quadrature rules.

This is the basic idea behind *Romberg integration*. By applying a quadrature rule repeatedly with more panels each time, we can eliminate the leading terms of the error expansion, and thereby obtain better and better approximations to $I(f)$.

9.5 Infinite Integrals and Singularities

If one or both end-points of the interval of integration $[a, b]$ are infinite, the integral is called infinite. We restrict the discussion of infinite integrals to the case

$$I(f) = \int_a^\infty f(x)\,dx,$$

sometimes called a semi-infinite integral, since only one end-point is infinite. Other cases can be handled similarly.

Assuming

$$I(f) = \int_a^\infty f(x)\,dx$$

exists, one way to approximate the integral is to *truncate* $I(f)$, and compute instead

$$\hat{I}(f) = \int_a^b f(x)\,dx,$$

for some sufficiently large b, by a standard quadrature rule $Q(f)$. The error in this approach is $I(f) - Q(f) = [I(f) - \hat{I}(f)] + [\hat{I}(f) - Q(f)]$. It is often possible to choose b so that

$$I(f) - \hat{I}(f) = \int_b^\infty f(x)\,dx$$

is small and to choose Q so that $\hat{I}(f) - Q(f)$ is also small.

$I(f)$ can also be approximated by first performing a change of variables to *transform* the infinite integral to a standard one. More specifically, let $x = g(t)$. Then

$$I(f) = \int_a^\infty f(x)\,dx = \int_{g^{-1}(a)}^{g^{-1}(\infty)} f(g(t))\cdot g'(t)\,dt,$$

where $g^{-1}(x)$ is the inverse of $g(x)$. If we can choose g so that $g^{-1}(a)$ and $g^{-1}(\infty)$ are both finite, then $I(f)$ is transformed to a finite integral which can be evaluated by a standard quadrature rule. However, this procedure may introduce singularities (discussed below). If so, it might not lead to a computationally easier problem to solve.

Infinite integrals can also be approximated by special quadrature formulas that are directly applicable to infinite intervals of integration. For further details concerning this approach, see an introductory numerical methods text such as (Conte and de Boor 1980; Dahlquist and Björck 1974; Johnson and Riess 1982; Stoer and Bulirsch 1980).

A singular integral

$$I(f) = \int_a^b f(x)\,dx$$

is one in which f is singular (i.e., becomes infinite) at some point in $[a, b]$. Singular and infinite integrals are closely related: a change of variables often transforms one into the other.

When computing singular integrals by a quadrature rule, the value of f might be required at or close to a point of singularity, and so the quadrature rule may be either inapplicable or inaccurate. It often happens that the singularity in f occurs at the end-point a or b, in which case, an open formula, such as a Gaussian rule, may be effective.

The performance of a quadrature rule applied to a singular integral might be improved by a change of variables. A transformation $x = g(t)$ which often helps to remove or lessen the effect of a singularity is $g(t) = b - (b-a)u^2(2u+3)$ for $u = (t-b)/(b-a)$.

9.6 Monte Carlo Methods

Monte Carlo methods are of a statistical nature. For the sake of simplicity, we briefly present them for one-dimensional integrals only, but they can be extended easily to multidimensional integrals and are most useful in this context.

To begin, choose n random points $\{U_i : i = 1, \ldots, n\} \subset [0, 1]$ and scale each U_i to $[a, b]$ by $u_i = a + U_i(b - a)$. Then,

$$Q_n(f) = (b - a)/n \sum_{i=1}^{n} f(u_i)$$

is a Monte Carlo approximation to

$$I(f) = \int_a^b f(x)\, dx.$$

If we consider $Q_n(f)$ to be a random variable, then it can be shown that its mean is $I(f)$ and its standard deviation is $|b - a|\rho(f)/\sqrt{n}$, where $\rho(f)$ is a constant that depends on f, but not n. Assuming that $Q_n(f)$ is close to being normally distributed, we are led to statistical statements about the error, such as $|Q_n(f) - I(f)| \leq 2|b - a|\rho(f)/\sqrt{n}$ nineteen times out of twenty.

Note the error bound above decreases like $1/\sqrt{n}$, much more slowly than the bounds for the standard compound quadrature rules given in Table 14. This suggests that Monte Carlo methods are not very effective for one-dimensional integrals of smooth functions. However, an error formula similar to that given above continues to hold for multi-dimensional integrals, while extensions of standard methods become increasingly less efficient as the dimension of the integral to be approximated increases. As a result, Monte Carlo methods are among the best schemes available for approximating high-dimensional integrals.

10 Ordinary Differential Equations

In this section, we consider numerical methods for the solution of ordinary differential equations (ODEs). We begin by introducing some simple schemes for the initial-value problem (IVP)

$$\begin{aligned} y'(x) &= f(x, y(x)) \qquad x \in [a, b] \\ y(a) &= y_0 \end{aligned} \tag{44}$$

where $y : \mathbb{R} \to \mathbb{R}^m$ and $f : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$. We assume throughout that the IVP (44) has a unique solution for $x \in [a, b]$. We also discuss more sophisticated adaptive methods and explain the terms *stiff* and *nonstiff* problems and how to choose methods appropriate for these two classes of problems. We then briefly consider the boundary-value problem (BVP)

$$\begin{aligned} y'(x) &= f(x, y(x)) \qquad x \in [a, b] \\ g(y(a), y(b)) &= 0 \end{aligned} \tag{45}$$

which we assume has a locally unique solution.

For both IVPs and BVPs, we consider systems of first-order ODEs only, since most commonly available

codes are for first-order systems and any higher-order ODE can be reduced to a system of first-order equations. However, using a method designed for higher-order equations directly may lead to a more efficient solution of the problem.

Because of space constraints, we do not discuss many important related problems such as differential-algebraic equations or delay differential equations. For a more comprehensive discussion of these and other related topics, see an advanced text such as (Ascher, Mattheij and Russell 1988; Butcher 1987; Hairer, Nørsett and Wanner 1987; Hairer and Wanner 1991; Lambert 1991; Shampine 1994).

10.1 Initial Value Problems

10.1.1 Two Simple Formulas

Most standard numerical methods for the IVP (44) start with the initial value y_0 at $x_0 = a$ and then compute approximations $y_n \approx y(x_n)$ for $n = 1, \ldots, N$ on a discrete grid $a = x_0 < x_1 < \cdots < x_N = b$. The distance between adjacent gridpoints, $h_n = x_{n+1} - x_n$, $n = 0, \ldots, N-1$, is referred to as the *stepsize* at step $n+1$. Schemes are often presented with a constant stepsize $h = h_n$ for all n, but this is generally not required. Moreover, as discussed below, variable-stepsize methods are often much more efficient.

Possibly the simplest numerical scheme for (44) is *Euler's method*, sometimes called the *forward Euler method*:

$$y_{n+1} = y_n + h_n f(x_n, y_n). \tag{46}$$

This formula is motivated from the observation that the true solution of a scalar IVP of the form (44) satisfies

$$\begin{aligned} y(x_{n+1}) &= y(x_n) + h_n y'(x_n) + \frac{h_n^2}{2} y''(\eta_n) \\ &= y(x_n) + h_n f(x_n, y(x_n)) + \frac{h_n^2}{2} y''(\eta_n) \end{aligned} \tag{47}$$

for some point $\eta_n \in [x_n, x_{n+1}]$, which follows from standard Taylor series theory. Thus, the true solution of the IVP (44) satisfies an equation that is very similar to (46). This argument can be extended easily to systems.

The approximations $y_n \approx y(x_n)$ are computed in the order $n = 1, \ldots, N$ using the formula (46). To be more specific, on the first step of Euler's method from x_0 to $x_1 = x_0 + h_0$, we substitute the initial value (x_0, y_0) into the right side of (46) to compute $y_1 \approx y(x_1)$. Thus, at the end of the first step, we have (x_1, y_1). On the second step of Euler's method from x_1 to $x_2 = x_1 + h_1$, we substitute (x_1, y_1) into the right side of (46) to compute $y_2 \approx y(x_2)$. The procedure continues in a similar way for $n = 2, \ldots, N$. Note that this evaluation process applies equally well to systems of equations (i.e., $m > 1$). In this case, h_n is a scalar, but y_{n+1}, y_n and $f(x_n, y_n)$ are all m-vectors.

Euler's method is an *explicit formula* in the sense that the evaluation process described above does not require the solution of any linear or nonlinear algebraic equations. The backward Euler formula

$$y_{n+1} = y_n + h_n f(x_{n+1}, y_{n+1}) \tag{48}$$

is a typical example of an *implicit formula*. It can be motivated from a Taylor series expansion of the true solution $y(x)$ of the IVP (44) about x_{n+1} similar to the expansion (47) above for $y(x)$ about x_n. Moreover, we again compute the approximations $y_n \approx y(x_n)$ in the order $n = 1, \ldots, N$. However, note that, on step $n+1$ from x_n to $x_{n+1} = x_n + h_n$, we start with (x_n, y_n) and must solve the equation (48) for y_{n+1}.

We will return to the question of how to solve for y_{n+1} shortly, but first we explain briefly in the next subsection why we may wish to use an implicit scheme (such as the backward Euler formula) rather than an explicit one (such as the forward Euler formula) even though the former clearly requires more work per step than the latter.

10.1.2 Stiff IVPs

Roughly speaking, a stiff IVP is one in which some terms in the solution decay rapidly with respect to the length of the integration, while others vary slowly on this time scale. To illustrate this concept, consider the linear constant coefficient problem $y' = Ay$, $y(0) = y_0$ for $x \in [0, 1]$, where

$$A = \frac{1}{2}\begin{pmatrix} -10^6 - 1 & 10^6 - 1 \\ 10^6 - 1 & -10^6 - 1 \end{pmatrix}, \qquad y_0 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}.$$

If we let $z = Py$, where

$$P = \frac{1}{2}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}, \qquad P^{-1} = \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}, \qquad D = PAP^{-1} = \begin{pmatrix} -10^6 & 0 \\ 0 & -1 \end{pmatrix},$$

then $z' = Py' = PAy = PAP^{-1}z = Dz$ and $z(0) = Py(0) = (1, 1)^T$. Therefore, $z(x) = (e^{-10^6 x}, e^{-x})^T$ and so $y(x) = P^{-1}z(x) = (e^{-x} + e^{-10^6 x}, e^{-x} - e^{-10^6 x})^T$. The term $e^{-10^6 x}$ that occurs in both $z(x)$ and $y(x)$ gives rise to an initial transient that decays rapidly with respect to the interval of integration $[0, 1]$, while the term e^{-x} is associated with a slowly varying smooth term on this scale. The fast and slow terms occur in different components of $z(x)$, but they are mixed in $y(x)$, which is often the case in practice. After the $e^{-10^6 x}$ term dies out, both components of $y(x)$ vary smoothly like e^{-x}.

If we apply the forward Euler formula to $y' = Ay$, we get $y_{n+1} = y_n + h_n A y_n = (I + h_n A)y_n$. If we multiply the last equation through by P and perform the change of variables $z_n = Py_n$, we get $z_{n+1} = (I + h_n D)z_n$. The two components of z_n satisfy $z_{n+1}^{(1)} = (1 - h_n 10^6) z_n^{(1)}$ and $z_{n+1}^{(2)} = (1 - h_n) z_n^{(2)}$. If we let $h_n > 2 \cdot 10^{-6}$ at any point during the integration, then $(1 - h_n 10^6) < -1$ and $z_n^{(1)}$ will grow in magnitude and oscillate in sign as n increases. Since $y_n = P^{-1} z_n$, this will cause both components of y_n to oscillate about e^{-x_n} with growing amplitude as n increases. Since this instability is undesirable, we must restrict $h_n < 2 \cdot 10^{-6}$ throughout the integration, even though, after the initial transient, this is likely a much smaller stepsize than would be required to integrate the slowly varying e^{-x} component of the solution accurately.

On the other hand, if we apply the backward Euler formula to $y' = Ay$, we get $y_{n+1} = y_n + h_n A y_{n+1}$ and so $y_{n+1} = (I - hA)^{-1} y_n$. If we multiply this equation through by P and perform the change of variables $z_n = Py_n$, we get $z_{n+1} = (I - h_n D)^{-1} z_n$. Therefore, $z_{n+1}^{(1)} = z_n^{(1)}/(1 + h_n 10^6)$ and $z_{n+1}^{(2)} = z_n^{(2)}/(1 + h_n)$.

Hence, no matter how large $h_n > 0$ is, $z_n^{(1)}$ decays as n increases. Consequently, after the initial transient, we can choose $h_n > 0$ to integrate $z_n^{(2)}$ accurately without fear of $z_n^{(1)}$ becoming unstable. Since $y_n = P^{-1}z_n$, the same conclusion applies to y_n.

The example above can be generalized to larger systems of equations $y' = Ay$. If A is diagonalizable, then the performance of the method on $y' = Ay$ can be deduced from its performance on the scalar test problems $y' = \lambda y$, where the λ's range over the eigenvalues of A. If the real part of each λ is negative, then $y(x) \to 0$ as $x \to \infty$. We would like the numerical solution to have the same behaviour without having to restrict h_n outside the transient region. Methods with this property are called *A-stable*. Generalizing the example above, it is easy to see that the backward Euler formula is A-stable, while the forward Euler formula is not.

The importance of the example above and the scalar test problem $y' = \lambda y$ in particular is that the performance of methods on these simple problems is indicative of their behaviour on more general nonlinear stiff problems. A nonrigourous, but intuitive, justification of this follows from the local linearization of $y' = f(x, y)$ at (x_n, y_n):

$$y'(x) \approx f(x_n, y_n) + f_x(x_n, y_n)(x - x_n) + f_y(x_n, y_n)(y - y_n)$$

where $f_y(x, y) = \partial f(x, y)/\partial y \in \mathbb{R}^{m \times m}$ is the Jacobian of f. This problem is usually stiff if

- some eigenvalue of $f_y(x_n, y_n)$ has a large negative real part with respect to the interval of integration,
- no eigenvalue of $f_y(x_n, y_n)$ has a large positive real part with respect to the interval of integration, and
- no eigenvalue of $f_y(x_n, y_n)$ has a large imaginary part unless it also has a relatively large negative real part.

An IVP which is not stiff is called *nonstiff*.

Stiff IVPs arise in many applications, such as chemical kinetics and electrical circuits. As noted earlier, they are characterized by components that vary on vastly different time scales: some terms in the solution decay rapidly to steady state while others vary slowly.

The observation above that the explicit forward Euler formula is not appropriate for a stiff problem, while the implicit backward Euler formula is, can be generalized. All commonly used formulas that are suitable for stiff problems are implicit in some sense.

See the survey article of Shampine and Gear (1979) or an advanced text such as (Butcher 1987; Hairer and Wanner 1991; Lambert 1991; Shampine 1994) for a more detailed discussion of stiffness.

10.1.3 Solving Implicit Equations

We return now to methods for solving for y_{n+1} in an implicit formula such as (48). One common approach is the predictor-corrector technique, which is just a fixed point iteration as described in §6.1. For this scheme (and most others), we need in initial approximation $y_{n+1}^{(0)}$ to y_{n+1}. This could be computed, for example, from the forward Euler formula or some other explicit scheme, or simply by taking $y_{n+1}^{(0)} = y_n$. In the

terminology of predictor-corrector techniques, the formula for computing $y_{n+1}^{(0)}$ is referred to as the *predictor* formula. For a predictor-corrector method based on the backward Euler formula (48), the corrector formula would be

$$y_{n+1}^{(l+1)} = y_n + h_n f(x_{n+1}, y_{n+1}^{(l)}), \qquad l = 0, 1, \ldots \tag{49}$$

We substitute $y_{n+1}^{(0)}$ into the right side of (49) to compute $y_{n+1}^{(1)}$, which we in turn substitute into the right side of (49) to compute $y_{n+1}^{(2)}$, and so on. It is easy to show that $y_{n+1}^{(l)} \to y_{n+1}$ as $l \to \infty$ if f satisfies the Lipschitz condition

$$\|f(x_{n+1}, y) - f(x_{n+1}, z)\| \leq L\|y - z\|$$

for some constant L and all y, z in a convex domain containing y_{n+1} and $y_{n+1}^{(l)}$ for $l = 0, 1, \ldots$ and $h_n L < 1$. In most codes, one or two corrections only are needed, since the initial guess $y_{n+1}^{(0)}$ is normally a good approximation to y_{n+1}. Consequently, this scheme is not much more expensive to implement than the explicit forward Euler formula. However, a predictor-corrector implementation of an A-stable method (such as the backward Euler formula) is not A-stable. On the contrary, because of the requirement $h_n L < 1$, it will suffer a stepsize restriction on a stiff problem similar to that of an explicit formula.

Alternatively, we could rewrite the backward Euler formula (48) as

$$F(y) = y - y_n - h_n f(x_{n+1}, y) = 0, \tag{50}$$

where we have replaced the unknown y_{n+1} by y, and then apply one of the other techniques described in §6 for finding roots of equations to compute the solution $y = y_{n+1}$ of (50). The most commonly used root finding technique in this context is Newton's method — or a variant of it. As noted in §6.7, for systems of equations, Newton's method takes the form

$$\left(I - h_n f_y(x_{n+1}, y_{n+1}^{(l)})\right) \Delta_l = y_n + h_n f(x_{n+1}, y_{n+1}^{(l)}) - y_{n+1}^{(l)} \tag{51}$$

$$y_{n+1}^{(l+1)} = y_{n+1}^{(l)} + \Delta_l \tag{52}$$

where $f_y(x, y) = \partial f(x, y)/\partial y \in \mathbb{R}^{m \times m}$ is the Jacobian of f. Note that we must solve a linear system of m equations in m unknowns to compute the Newton update vector Δ_l in (51). Typically, Gaussian elimination with partial pivoting (see §2.5) is used to solve such linear systems. A band or sparse solver (see §2.7) may dramatically decrease the cost of solving (51) if $I - h_n f_y(x_{n+1}, y_{n+1}^{(l)})$ is large and sparse. Similarly, iterative methods, such as the preconditioned conjugate gradient method (see §3.2) may significantly reduce the cost of solving some large sparse problems. See §13 for a discussion of sources of high-quality numerical software, including stiff ODE solvers that incorporate sparse and iterative linear equation solvers.

The computational work required to solve (51) can be decreased significantly by using a chord Newton method, often called a simplified Newton method, that holds the Newton iteration matrix $I - h_n f_y(x_{n+1}, y_{n+1}^{(l)})$ constant over several iterations and possibly several steps of the integration, thus avoiding the necessity to factor the Newton iteration matrix on each iteration (see §6.8). However, even with this savings, the cost per step of the Newton iteration may be much larger than a predictor-corrector method. However, it has

the advantage for formulas appropriate for stiff problems that it avoids the stepsize restriction associated with the predictor-corrector technique or explicit formulas. Thus, even though the Newton iteration might make the scheme much more expensive per step, the stepsizes that can be used might be so much larger that the total cost of the integration is significantly less. Finally note that, like predictor-corrector methods, an initial guess for $y_{n+1}^{(0)}$ is required. It can be computed by the techniques described above.

10.1.4 Higher Order Formulas

A numerical method for ODEs is said to be *of order p* or p^{th} *order* or p^{th} *order convergent* if $y_n = y(x_n) + O(h^p)$ for some integer $p \geq 1$, where $O(h^p)$ is any quantity (in this case, the global error) that can be bounded by h^p times a constant that is independent of h, but which may depend on the IVP and the numerical method. Most standard texts on the numerical solution of ODEs show that both the forward and backward Euler methods are first order convergent.

Higher order methods are frequently used in practice because they offer the potential of significantly reducing the computational work required to generate an accurate solution to the IVP (44). To get an intuitive feeling for this, suppose that the length of integration $b - a = 1$, we use a constant stepsize h throughout the numerical integration, the global error for a p^{th} order method satisfies $y_n - y(x_n) = h^p$, and we require that this error be of size 10^{-10}. Under these assumptions, the optimal stepsize for the method is $h = 10^{-10/p}$ and the resulting number of steps needed to integrate from a to b is $N = 10^{+10/p}$. To be more specific, for $p = 1, 2, 5, 10$, the number of steps required is $N = 10^{10}, 10^5, 10^2, 10^1$, respectively. Thus, even though a higher order method may require more work per step than a lower order scheme, the dramatic reduction in the number of steps required frequently makes a higher order method much more efficient than a lower order one — particularly for problems with stringent error tolerances.

Two common second-order formulas are the *trapezoidal rule*

$$y_{n+1} = y_n + \frac{h_n}{2}[f(x_n, y_n) + f(x_{n+1}, y_{n+1})] \tag{53}$$

and the *implicit midpoint rule*

$$y_{n+1} = y_n + f(x_n + h_n/2, [y_n + y_{n+1}]/2) \tag{54}$$

each of which is implicit, since one clearly needs to solve for y_{n+1}. Note that neither formula requires much more work per step than the backward Euler formula. Moreover, both formulas are A-stable and effective for solving stiff problems at relaxed error tolerances.

10.1.5 Runge-Kutta Formulas

Runge-Kutta (RK) formulas are a general class of methods containing many higher-order schemes. The general form of an s-stage RK formula is

$$\begin{aligned} k_i &= f(x_n + c_i h_n, y_n + h_n \textstyle\sum_{j=1}^{s} a_{ij} k_j), \qquad i = 1, \ldots, s, \\ y_{n+1} &= y_n + h_n \textstyle\sum_{i=1}^{s} b_i k_i. \end{aligned} \tag{55}$$

That is, we must first compute the s function values k_i and then form a weighted average of the k_i's to compute y_{n+1} from y_n.

RK formulas are 1-step schemes in the sense that all the information required to compute y_{n+1} from y_n is generated on the current step from x_n to x_{n+1}. That is, unlike multistep formulas discussed in the next subsection, a RK formula does not require any information from past steps.

If the stages of the RK formula can be ordered so that $a_{ij} = 0$ for all $j \geq i$, then the formula (55) is explicit in the sense that the k_i's can be computed in the order $i = 1, \ldots, s$ without having to solve any linear or nonlinear equations. In what follows, we assume that the formula has been so ordered if possible. If the RK formula (55) is not explicit, then it is implicit and at least one linear or nonlinear equation must be solved to compute the k_i's.

The coefficients of a RK formula are frequently displayed in a tableau as

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

If all the elements in the tableau on the diagonal and above it are zero, then the associated formula is explicit.

All the methods considered so far are in fact RK formulas. The RK tableaux for the forward Euler formula, backward Euler formula, implicit midpoint rule and trapezoidal rule are listed below from left to right, respectively.

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \qquad \begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \qquad \begin{array}{c|c} 1/2 & 1/2 \\ \hline & 1 \end{array} \qquad \begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1/2 & 1/2 \\ \hline & 1/2 & 1/2 \end{array}$$

Note that the first three are 1-stage RK formulas and the final one, the trapezoidal rule, is a 2-stage RK formula. The tableau for the classical 4-stage 4^{th}-order explicit RK formula is

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

This formula has been widely used since it was published by Kutta in 1901. In the days of hand calculation, the zero coefficients below the diagonal were a distinct benefit, but this is no longer a significant advantage on a modern computer. Moreover, there are now many better formulas, both of order 4 and of higher order. The interested reader should consult an advanced text such as (Butcher 1987; Hairer, Nørsett and Wanner 1987; Hairer and Wanner 1991; Lambert 1991; Shampine 1994) for further details.

As the sample formulas above suggest, a high-order RK formula requires more stages than a low-order one. The minimum number of stages that an explicit RK formula requires to attain orders 1 to 8 are listed below.

Order	1	2	3	4	5	6	7	8
Stages	1	2	3	4	6	7	9	11

On the other hand, implicit s-stage RK formulas of order $2s$ exist for all $s \geq 1$. Moreover, it can be shown that this is the maximal order possible.

Explicit RK formulas are frequently used to solve non-stiff IVPs. Some implicit RK formulas are A-stable (or nearly so) and are suitable for solving stiff IVPs. Formulas with four or fewer stages are quite effective for problems with relaxed error tolerances, while formulas with five or more stages are suitable for problems with more stringent accuracy requirements. Since RK formulas are 1-step schemes, unlike linear multistep formulas (LMFs) discussed in the next subsection, RK formulas are more suitable than LMFs for problems that require rapid changes in stepsize, such as problems with discontinuities. See §13 for a discussion of sources of high-quality numerical software, including routines based on RK formulas.

10.1.6 Linear Multistep Formulas

Linear multistep formulas (LMFs) can be written in the form

$$y_{n+1} = \sum_{i=1}^{k} \alpha_i y_{n+1-i} + h \sum_{i=0}^{k} \beta_i f(x_{n+1-i}, y_{n+1-i}) \tag{56}$$

where we assume at least one of α_k or β_k is nonzero (otherwise we can reduce k in (56)). Formula (56) is in fact a k-step method, since it uses values over k steps to compute y_{n+1}. Therefore, we assume that, at the start of step $n+1$, we have $y_{n+1-k}, \ldots, y_n$ and our task is to compute y_{n+1} by (56). In this case, if $\beta_0 = 0$, then the $f(x_{n+1}, y_{n+1})$ term can be dropped from the right side of (56), and so we can evaluate the right side of (56) to compute y_{n+1} without having to solve any linear or nonlinear equations. That is, if $\beta_0 = 0$, the formula is explicit. On the other hand, if $\beta_0 \neq 0$, then y_{n+1} occurs on both sides of (56), and so a linear or nonlinear equation must be solved to compute y_{n+1}. Therefore, the formula is implicit. Depending on the context, a predictor-corrector method or some variant of Newton's method is typically used to solve for y_{n+1}.

Of course, at the start of the integration, $n = 0$ and $y_{1-k}, \ldots, y_{-1}$ are typically not available for $k > 1$. One solution to this problem is to compute $y_1, \ldots, y_{k-1}$ by a 1-step method (such as a RK formula) of the same order as the k-step LMF and then start using the k-step LMF at step k. Alternatively, we could use a 1-step LMF on step 1, a 2-step LMF on step 2, and so on, until we reach step k, after which we can use (56)

on that step and all subsequent steps. Most LMF codes employ the latter strategy and adjust the stepsize so that the accuracy obtained by the formulas with smaller k is comparable to that obtained by formulas with larger k.

If we ignore stability, then it is possible to obtain a k-step LMF of order $2k$ for any $k \geq 1$. Moreover, it is easy to show that this is the maximal order possible. However, these maximal order formulas are unstable for $k \geq 3$ and so are not useful in practice. It can be shown that, for any $k \geq 1$, the maximal order of a stable k-step LMF is $k+1$ if k is odd and $k+2$ if k is even.

We have presented the LMFs in this section for fixed stepsize only, since variable-stepsize formulas are considerably more complicated. However, a variable-stepsize variable-order scheme may be far more efficient in practice. Such schemes are discussed in advanced texts on the numerical solution of ODEs, such as (Hairer, Nørsett and Wanner 1987; Hairer and Wanner 1991; Lambert 1991; Shampine 1994), but not usually in introductory numerical methods books.

10.1.7 Adams Formulas

Adams formulas are a subclass of LMFs that have the form

$$y_{n+1} = y_n + h \sum_{i=0}^{k} \beta_i f(x_{n+1-i}, y_{n+1-i}). \tag{57}$$

In the explicit Adams-Bashforth formulas, $\beta_0 = 0$ and the remaining k β_i's are chosen to obtain the maximal possible order k. In the implicit Adams-Moulton formulas, $\beta_0 \neq 0$ and the $k+1$ β_i's are chosen to obtain the maximal possible order $k+1$. These coefficients are listed in most advanced texts on numerical methods for ODEs and in many introductory numerical methods books. It turns out that the forward Euler formula is the 1-step Adams-Bashforth formula and the trapezoidal rule is the 1-step Adams-Moulton formula. Moreover, note that the order of the Adams-Moulton formulas is optimal for k odd and nearly optimal for k even.

The implicit Adams-Moulton formulas have somewhat better numerical characteristics than the explicit Adams-Bashforth formulas. Consequently, Adams formulas are usually implemented in a predictor-corrector fashion, with the k or $k+1$ step Adams-Bashforth formula used for the predictor and a k step Adams-Moulton formula used for the corrector. Adams predictor-corrector schemes are the basis for several very effective variable-stepsize variable-order codes for nonstiff IVPs. See §13 for a discussion of sources of high-quality numerical software, including routines based on Adams formulas.

10.1.8 Backward Differentiation Formulas

Backward Differentiation Formulas (BDFs), sometimes called *Gear formulas*, are another subclass of LMFs that have the form

$$y_{n+1} = \sum_{i=1}^{k} \alpha_i y_{n+1-i} + h\beta_0 f(x_{n+1}, y_{n+1}) \tag{58}$$

where $\beta_0 \neq 0$ and so the BDFs are implicit. The $k+1$ coefficients of a k-step BDF are chosen to obtain the maximal possibly order k. However, the BDFs are stable for $1 \leq k \leq 6$ only. They are A-stable for $k = 1$

and 2 and nearly A-stable for $k = 3$, 4 and 5, with the loss of A-stability increasing with k. For $k = 6$ the loss of A-stability increases to such an extent that this formula is frequently excluded from use.

The coefficients for the BDFs are listed in most advanced texts on numerical methods for ODEs and in many introductory numerical methods books. It turns out that the backward Euler formula is the 1-step BDF.

Because the BDFs are usually used to solve stiff problems, the implicit equation is normally solved by Newton's method or some variant of this root finding scheme.

BDFs are the basis for several very effective variable-stepsize variable-order codes for stiff IVPs. See §13 for a discussion of sources of high-quality numerical software, including routines based on BDFs.

10.1.9 Other Methods

Taylor series methods and extrapolation schemes are two other classes of formulas that are sometimes used in practice, but much less frequently than Runge-Kutta or linear multistep formulas. See an advanced text such as (Butcher 1987; Hairer, Nørsett and Wanner 1987; Hairer and Wanner 1991; Lambert 1991; Shampine 1994) for a discussion of these and other classes of methods.

10.1.10 Adaptive Methods

Most good programs for the numerical solution of ODEs vary their stepsize — and possibly their order — in an attempt to solve the problem as efficiently as possible subject to a user specified error tolerance, *tol*. The error that is controlled is usually the local error on each step, rather than the global error $y_n - y(x_n)$ that the user might at first expect. However, in most good programs the global error is at least roughly proportional to *tol*, so reducing *tol* usually reduces the global error. A few codes report an estimate of the global error as well. If such an estimate is available, it is often optional, since estimating the global error frequently increases the cost of the integration significantly.

A useful way to interpret *tol* and the associated local error is in the *backward error* sense. (See §2.8 for a discussion of backward error analysis in the context of solving linear algebraic systems $Ax = b$.) When called to solve the IVP (44), many good programs generate a numerical solution that is the exact solution of the slightly perturbed problem

$$z'(x) = f(x, z(x)) + \delta(x), \qquad z(a) = y_0, \tag{59}$$

where $\|\delta(x)\| \lessapprox tol$. A few codes compute $\delta(x)$ explicitly and attempt to ensure that it is bounded by *tol*, but most satisfy (59) indirectly (some less reliably than others) by controlling some measure of the local error. For a more complete discussion of global errors, local errors and their relationship to the perturbed equation (59), see an advanced text such as (Butcher 1987; Hairer, Nørsett and Wanner 1987; Hairer and Wanner 1991; Lambert 1991; Shampine 1994).

We believe that this backward error approach is often the most natural way to view the error in the numerical integration of an IVP. In many practical problems, we know $f(x, y)$ approximately only, possibly

because of measurement errors or neglected terms in the model. Therefore, the true solution of the system satisfies an equation of the form (59), where in this case $\delta(x)$ is the error in the model. So, any solution to an IVP of the form (59) may be equally good provided $\|\delta(x)\|$ is less than the error in the model.

Programs for the numerical solution of ODEs often contain many other useful features. For example, some routines for nonstiff IVPs warn the user if the problem is stiff, while others automatically switch between stiff and nonstiff methods depending on the characteristics of the problem. Some programs contain sophisticated strategies to integrate problems with discontinuities in f or its derivatives much more efficiently and reliably than programs that do not attempt to detect discontinuities. Also, some programs return an interpolant for the numerical solution or allow the user to evaluate the numerical solution at very closely spaced points more efficiently than if the integration method itself produced all these output points. This can greatly increase the efficiency of codes when used to produce graphical output or to detect when the numerical solution satisfies some condition (such as $y(x) = c$ for some constant c).

See §13 for a discussion of sources of high-quality numerical software, including routines for the numerical solution of IVPs for ODEs.

10.2 Boundary Value Problems

10.2.1 Shooting Methods

Shooting is conceptionally one of the simplest numerical techniques for solving the boundary-value problem (BVP) (45). In its simplest form, often called *simple shooting*, we guess an initial condition $y(a) = y_0$ for the IVP (44) for the same ODE as the BVP (45), solve the IVP (44), and test whether the boundary condition $g(y_0, y(b; a, y_0)) = 0$ is satisfied, or nearly so, where $y(b; a, y_0)$ is the solution at $x = b$ of the IVP (44) with the initial condition $y(a) = y_0$. In most cases, the first guess for the initial condition $y(a) = y_0$ does not yield a $g(y_0, y(b; a, y_0))$ that is close enough to 0. So we must apply some root finding technique to adjust the initial condition $y(a) = y_0$ until $g(y_0, y(b; a, y_0)) = 0$ is satisfied, or nearly so, assuming that there is a solution to the BVP.

Each time we adjust the initial condition, we must solve the IVP (44) again with the new initial condition $y(a) = y_0^{(l)}$ to compute $y(b; a, y_0^{(l)})$ and then $g(y_0^{(l)}, y(b; a, y_0^{(l)}))$. For a scalar ODE ($m = 1$), we could try a simple technique such as bisection (see §6.4) to solve $g(y_0, y(b; a, y_0)) = 0$, but this converges slowly and so requires many solutions of the IVP (44) with different initial conditions $y_0^{(l)}$. Moreover, bisection is not applicable to systems of ODEs ($m > 1$).

The usual approach is to apply a variant of Newton's method (see §6.7) to solve $g(y_0, y(b; a, y_0)) = 0$. However, this requires that we compute an approximation to the Newton iteration matrix

$$\frac{dg(y_0^{(l)}, y(b; a, y_0^{(l)}))}{dy_0} = \frac{\partial g(y_0^{(l)}, y(b; a, y_0^{(l)}))}{\partial y_a} + \frac{\partial g(y_0^{(l)}, y(b; a, y_0^{(l)}))}{\partial y_b} \frac{\partial y(b; a, y_0^{(l)})}{\partial y_0}$$

where $\partial g(y_0^{(l)}, y(b; a, y_0^{(l)}))/\partial y_a$ is the partial derivative of g with respect to its first argument, $\partial g(y_0^{(l)}, y(b; a, y_0^{(l)}))/\partial y_b$ is the partial derivative of g with respect to its second argument, and $\partial y(b; a, y_0^{(l)})/\partial y_0$ is the partial derivation

of $y(b;a,y_0^{(l)})$ with respect to the initial condition $y(a) = y_0^{(l)}$. It can be shown that $\partial y(b;a,y_0^{(l)})/\partial y_0 = Y_l(b)$ for $Y_l : \mathbb{R} \to \mathbb{R}^{m\times m}$ the solution of the *variational equation*

$$Y_l'(x) = f_y(x, y_l(x))Y_l(x), \qquad Y_l(a) = I, \tag{60}$$

where $y_l(x)$ is the solution of the associated IVP (44) with initial condition $y(a) = y_0^{(l)}$ and $f_y(x,y) = \partial f(x,y)/\partial y \in \mathbb{R}^{m\times m}$ is the Jacobian of f. Therefore, on each iteration of Newton's method, we must solve the IVP (44) with initial condition $y(a) = y_0^{(l)}$ for $y_l(x)$ as well as the variational equation (60) associated with $y_l(x)$. Since it may take many iterations before we find a $y_0^{(l)}$ for which $g(y_0^{(l)}, y(b;a,y_0^{(l)})$ is sufficiently close to 0, this is often a computationally expensive process.

Moreover, the associated IVP (44) may be unstable even though the BVP (45) is stable. As a result, simple shooting may break down or perform poorly. One way around this difficulty is to employ *multiple shooting*. In this scheme, we choose $N+1$ shooting points $\{x_0 : i = 0,\ldots,N\}$ satisfying $a = x_0 < x_1 < \cdots < x_{N-1} < x_N = b$, guess at N initial conditions s_i, $i = 0,\ldots,N-1$, and solve the N IVPs

$$\begin{aligned} y_i' &= f(x,y_i), \qquad x \in [x_i, x_{i+1}] \qquad i = 0,\ldots,N-1 \\ y_i(x_i) &= s_i. \end{aligned} \tag{61}$$

These IVPs are completely independent and so could be integrated simultaneously. Hence, this scheme is often called *parallel shooting*.

We need to adjust the initial conditions s_i, $i = 0,\ldots,N-1$, so that

$$y_i(x_{i+1}) = s_{i+1}, \qquad i = 0,\ldots,N-2 \tag{62}$$

$$g(s_0, y_N(b)) = 0 \tag{63}$$

where the first set of conditions (62) ensures $y_i(x_{i+1}) = y_{i+1}(x_{i+1})$ at the $N-1$ interior shooting points $x_1,\ldots,x_{N-1}$, thus allowing us to patch the functions $y_i(x)$ together into a continuous function $y(x)$ on $[a,b]$, and the second condition (63) enforces the boundary condition for the BVP (45).

A variant of Newton's method (see §6.7) is usually used to solve (62)–(63). The solution process is similar to, but somewhat more complicated than, that described above for simple shooting. It should be noted that the linear systems associated with Newton's method for (62)–(63) have a very special structure than can be exploited to great computational advantage. See an advanced text such as (Ascher, Mattheij and Russell 1988) for details.

Both simple and multiple shooting simplify significantly when applied to a linear ODE $y' = A(x)y + b(x)$. Newton's method converges in one iteration and the resulting scheme is equivalent to what is frequently called the method of *superposition*. If the boundary conditions are separated, this scheme simplifies still further. See an advanced text such as (Ascher, Mattheij and Russell 1988) for details.

Good shooting programs contain heuristics for choosing the shooting points and adjusting the tolerance for the IVP solver in an attempt to solve the BVP to within a user specified tolerance. They also contain

many other components, similar to those described in §10.1.10 for IVPs. See an advanced text such as (Ascher, Mattheij and Russell 1988) for a discussion of these features and §13 for a discussion of sources of high-quality numerical software, including methods for the numerical solution of BVPs for ODEs.

10.2.2 1-Step Methods

It is common to apply a 1-step method, such as a Runge-Kutta (RK) formula, to solve the BVP (45). Since a collocation method applied to an ODE often reduces to a RK formula, this class of methods is broader than it might at first appear.

To simplify the discussion, assume that the 1-step method can be written in the form

$$y_{n+1} = y_n + h_n \phi(x_n, y_n, h_n) \tag{64}$$

where $y_n \approx y(x_n)$, $h_n = x_{n+1} - x_n$ and $a = x_0 < x_1 < \cdots < x_N = b$. Note that the RK formula (55) is of this form with

$$\begin{aligned} \phi(x_n, y_n, h_n) &= \textstyle\sum_{i=1}^{s} b_i k_i \\ k_i &= f(x_n + c_i h_n, y_n + h_n \textstyle\sum_{j=1}^{s} a_{ij} k_j) \end{aligned} \tag{65}$$

To apply the 1-step formula (64) to the BVP (45), we simply combine the equations (64) together with the boundary conditions to get a large systems of equations

$$\Phi(y_0, \ldots, y_N) = \left\{ \begin{array}{c} y_{n+1} - y_n - h_n \phi(x_n, y_n, h_n), \quad n = 0, \ldots, N-1, \\ g(y_0, y_N) \end{array} \right\} = 0. \tag{66}$$

It is usual to apply a variant of Newton's method (see §6.7) to solve (66). As for shooting, the main difficulty here is to compute the $(N+1)m \times (N+1)m$ Newton iteration matrix $\partial\Phi(y_0, \ldots, y_N)/\partial(y_0, \ldots, y_N)$ and solve the associated linear system for the update to the approximate solution $y_0^{(l)}, \ldots, y_N^{(l)}$ to (66). See an advanced text such as (Ascher, Mattheij and Russell 1988) for a more complete discussion of this important point.

Good BVP codes contain heuristics for choosing the gridpoints to solve the BVP to within a user specified tolerance. They also contain many other components, similar to those described in §10.1.10 for IVPs. See an advanced text such as (Ascher, Mattheij and Russell 1988) for a discussion of these features and §13 for a discussion of sources of high-quality numerical software, including methods for the numerical solution of BVPs for ODEs.

10.2.3 Other Methods

There are several other classes of numerical methods for BVPs for ODEs. Some of these are discussed in §11 as numerical methods for BVPs for partial differential equations. An important class of methods, not discussed there, are *defect correction* schemes, including *deferred correction* as a special case. The basic idea behind these schemes is to apply a simple technique, possibly in the class discussed in the last subsection, and then estimate the *defect* or truncation error in the discretization and solve a related problem again with

the same simple technique in an attempt to eliminate the error. See an advanced text such as (Ascher, Mattheij and Russell 1988) for further details.

11 Partial Differential Equations

A partial differential equation (PDE) is an equation in which the partial derivative of some order of the unknown function w.r.t. some independent variable occurs. For example,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = g(x, y) \tag{67}$$

is a PDE, where $u(x, y)$ is an unknown function, $\partial^2 u/\partial x^2$ and $\partial^2 u/\partial y^2$ denote the partial second derivatives of u w.r.t. x and y, respectively, (often denoted by u_{xx} and u_{yy}, respectively), and $g(x, y)$ is a given function. The terms that involve u and its derivatives define the (partial differential) *operator* L, where, for example, $L = \partial^2/\partial x^2 + \partial^2/\partial y^2$ in (67), and the rest of the terms (usually the right side of the equation) form the *source term*.

11.1 Classes of Problems and PDEs

PDEs describe many important physical and technological phenomena. These phenomena can be divided into two basic types, which in turn are associated with two basic classes of problems for PDEs.

1. **Equilibrium phenomena, elliptic PDEs, boundary value problems.** In steady state phenomena, the equilibrium configuration u often satisfies

$$Lu = g \quad \text{in} \quad \Omega \tag{68}$$

$$Bu = \gamma \quad \text{on} \quad \partial\Omega \tag{69}$$

   where Ω is a spatial N-dimensional domain, $\partial\Omega$ is the boundary of Ω, u is the unknown function of N variables, g and γ are known functions of N variables and L and B are partial differential operators. Such problems are called *boundary value problems* (BVPs). Often L is an elliptic operator. Equation (69) is frequently referred to as the *boundary condition* (BC). The definition of an elliptic operator in the general case is beyond the scope of this article, but some typical examples are given below.

2. **Propagation phenomena, parabolic and hyperbolic PDEs, initial value problems.** In phenomena of a transient nature, the initial state is often given and we wish to predict the subsequent behaviour. The function u at some point $t \in (0, T)$ frequently satisfies

$$Lu = g \quad \text{in} \quad \Omega \times (0, T) \tag{70}$$

$$Bu = \gamma \quad \text{on} \quad \partial\Omega \times (0, T) \tag{71}$$

while the initial configuration satisfies

$$Iu = g_0 \quad \text{in} \quad \Omega \cup \partial\Omega \tag{72}$$

where $(0, T)$ is the time interval of interest, Ω is a spatial N-dimensional domain, $\partial\Omega$ is the boundary of Ω, u is the unknown function of N spatial variables and one time variable t, g and γ are known functions of N spatial variables and t, g_0 is a known function of N spatial variables, and L, B and I are partial differential operators. Such problems are called *initial value problems* (IVPs). L is often either a parabolic or a hyperbolic operator (see below). Equation (72) is often referred to as an *initial condition* (IC).

11.1.1 Some Definitions

The *dimension* of a PDE is the number of independent variables in the PDE. The *order* of a PDE is the order of the highest derivative of the unknown function occurring in the PDE. A PDE is called *linear* if there are no nonlinear terms in the equation involving the unknown function or its derivatives; otherwise it is called *nonlinear*.

For example,

$$\sum_{i=1}^{N}\sum_{j=1}^{N} a_{ij}(x)\frac{\partial^2 u}{\partial x_i \partial x_j} + \sum_{j=1}^{N} b_j(x)\frac{\partial u}{\partial x_j} + c(x)u = d(x) \tag{73}$$

is N-dimensional, second order and linear, where $x = (x_1, \ldots, x_N)$ is an N-dimensional vector of independent variables, $u(x)$ is the unknown function and $\{a_{ij} : i = 1, \ldots, N,\ j = 1, \ldots, N\}$, $\{b_j : j = 1, \ldots, N\}$, c and d are given functions of x. If any of the functions a_{ij}, b_j or c depend on u or its derivatives, of if d is nonlinear in u or its derivatives, then the PDE is nonlinear.

A linear operator L is called *positive definite* if $(Lu, u) > 0$ for all $u \neq 0$, where $(\cdot, \cdot)$ denotes an inner product (see §8.7). In addition, L is called *self-adjoint* if $(Lu, v) = (u, Lv)$ for all u and v in the associated function space. For example, the two-dimensional second-order linear PDE

$$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + fu = g \tag{74}$$

is self-adjoint if $d(x, y) = \partial a/\partial x$, $e(x, y) = \partial c/\partial y$ and $b(x, y) = 0$.

Consider the linear differential equation $Lu = g$, with L self-adjoint and positive definite. Consider also the quadratic functional $F(u)$ defined by $F(u) = (Lu, u) - 2(u, g)$. The *minimum functional theorem* states that the solution of the differential equation $Lu = g$ coincides with the function u that minimizes $F(u)$. In general, when a solution to a differential equation corresponds to an extremum of a related functional, we have a *variational principle*. Numerical methods that construct approximations to the solution of a differential equation by using such a relationship are called *variational methods*, *Ritz methods*, *Rayleigh-Ritz methods* or *energy methods*. The latter term comes from the observation that many variational methods are based on the physical principle of energy minimization.

The two-dimensional second-order linear PDE (74) is *elliptic*, *parabolic* or *hyperbolic* if $D = b^2 - 4ac < 0$,

$= 0$ or > 0, respectively. For the definitions of these terms in the general case, the reader is referred to any introductory PDE book, such as (Ames 1992; Celia and Gray 1992; Hall and Porsching 1990). Typical examples of elliptic, parabolic and hyperbolic PDEs are

- Laplace's equation, $u_{x_1x_1} + u_{x_2x_2} + \cdots + u_{x_{N-1}x_{N-1}} + u_{x_Nx_N} = 0$, which is elliptic,
- the heat equation, $u_{x_1x_1} + u_{x_2x_2} + \cdots + u_{x_{N-1}x_{N-1}} - u_{x_N} = 0$, which is parabolic, and
- the wave equation, $u_{x_1x_1} + u_{x_2x_2} + \cdots + u_{x_{N-1}x_{N-1}} - u_{x_Nx_N} = 0$, which is hyperbolic.

Two other classical elliptic PDEs (given in two dimensions) are

- Poisson's equation, $u_{xx} + u_{yy} = f(x, y)$, and
- the Helmholtz equation, $u_{xx} + u_{yy} + \kappa u = f(x, y)$, where κ is a constant.

The *normal derivative* of a surface $u(x, y)$ is the rate of change of u along the direction of the outward normal (i.e., the direction perpendicular to the surface). Let α be the angle which the direction of the outward normal makes with the x-axis at a point (x, y) on u. Then the normal derivative $\partial u/\partial n$ of u (often denoted by u_n) at the point (x, y) is $u_n = u_x \cos\alpha + u_y \sin\alpha$. The normal derivative can also be written as the inner product of the gradient of u, ∇u, with the unit outward normal vector, n. That is, $u_n = \nabla u \cdot n$. This definition of the normal derivative for two dimensions can be generalized easily to N dimensions.

11.1.2 Boundary Conditions

We now list some common types of boundary conditions (BCs) corresponding to the partial differential operator B in (69) or (71).

- *Dirichlet*: $Bu = u$.
- *Neumann*: $Bu = u_n$.
- *General* (for second order PDEs): $Bu = \alpha(x)u + \beta(x)u_n$.
- *Mixed* (for second order PDEs): often, on parts of the boundary we have Dirichlet BCs and on the other parts Neumann ones. (Also, the term "mixed" may sometimes refer to more general types of BCs.)
- *Essential*: for PDEs of order $2m$, essential BCs involve u and its derivatives of order up to $m - 1$.
- *Natural*: for PDEs of order $2m$,, natural BCs involve the derivatives of u from order m to $2m - 1$.

For further reading on the classification of PDE problems, operators and boundary conditions, see (Ames 1992; Celia and Gray 1992; Hall and Porsching 1990).

11.2 Classes of Numerical Methods for PDEs

The two most commonly used methods for approximating the solution of PDEs are described briefly below.

Finite Difference Methods (FDMs). The main steps of a FDM are the following.

1. Choose a finite difference (FD) approximation of the derivatives involved in the PDE, BCs and ICs. The result is a discretized PDE, BCs and ICs.

2. Choose a set of n data points in the domain and on the boundary, on which the discretized PDE, BCs and ICs must be satisfied. The result is a set of n equations w.r.t. the approximate values of u at the n data points.

3. Write the n equations of step 2 as a system and solve the system (discrete model). (If the PDE is linear, the system will usually be linear.) The solution is the approximate value of u at each of the n data points.

4. Evaluate the approximation to u at some point(s) of the domain (if needed).

Finite Element Methods (FEMs). The main steps of a FEM are the following.

1. Choose a finite element (FE) space, say n-dimensional, which the approximation u_Δ is constrained to belong to, and a set of basis functions that span the space, say $\{\phi_i : i = 1, \ldots, n\}$. Then write

$$u_\Delta(x) = \sum_{i=1}^{n} \alpha_i \phi_i(x).$$

   The unknown scalars α_i, $i = 1, \ldots, n$, are often called the *degrees of freedom* (DOF), or *coefficients*, of the FE representation of u_Δ.

2. Choose a set of n conditions that the approximation u_Δ must satisfy. The result is a set of n equations w.r.t. the n coefficients of u_Δ.

3. Write the n equations of step 2 as a system and solve the system (discrete model). (If the PDE is linear, the system will usually be linear.) The solution is the vector of coefficients of u_Δ.

4. Evaluate the approximation to u at some point(s) of the domain.

11.2.1 Analysis of Numerical Methods for PDEs

Some common techniques used to analyze numerical methods for PDEs are discussed below. The analysis can be used to evaluate a method w.r.t. some chosen criteria or measures. In the discussion below, we use u_Δ to denote the approximation to u computed by the method.

Convergence analysis (for BVPs and IVPs). We study the behaviour of the error $u - u_\Delta$ as n increases. Assuming $\|u - u_\Delta\| \to 0$ as $n \to \infty$, we can write $\|u - u_\Delta\| \leq C(1/n)^\alpha$ for some constants C and α. The largest constant α for which this inequality holds is called the *order of convergence* of the method. As a first rough measure, the larger the α the better the method, as $(1/n)^\alpha$ will converge to 0 faster as $n \to \infty$ for larger α. To estimate the order of convergence of a method experimentally, we often devise

PDE problems with known solutions and then solve them using the PDE method under investigation, first using n DOF, then $2n$ DOF, etc. We then plot $\|u - u_\Delta\|$ versus n on a log-log scale. The slope of the plotted line is an approximation to the order of convergence of the method.

Stability analysis (for IVPs). We study the behaviour of the error $u - u_\Delta$ as a function of t for increasing t. We often say that a method is *stable* if $\|u - u_\Delta\|$ remains bounded as $t \to \infty$. Otherwise, it is called *unstable*. Or, we study how the error at some point in time propagates to the next point in time. In a stable method, the error is not amplified.

Time (computational) complexity analysis. We study the time that the method takes to compute the approximate solution to the PDE as a function of the n DOF. The time is usually proportional to the number of floating-point operations, although it also depends on the implementation and the hardware (computer) used. The most time consuming part of a FDM or FEM is usually the third step (solution of the system), while the second most time consuming part is usually the second step (generation of the system). For FDMs, the fourth step can also be time consuming, particularly if the value of the approximation at arbitrary points of the domain is required, since this computation requires interpolation, often using piecewise polynomials (PPs) or splines. The data to be interpolated are the approximate values of u at the gridpoints. Interpolation is not required in step 4 of a FEM, since the approximate solution can be evaluated at any point of the domain by the formula

$$u_\Delta(x) = \sum_{i=1}^{n} \alpha_i \phi_i(x).$$

By studying the particular implementation of a method, we are usually able to derive an approximate formula, such as time $\approx Kn^\beta$, for some constant K, or time $= O(n^\beta)$, relating the computational complexity to n. The smaller the β, the faster the method, and, among methods with the same β, the smaller the K, the faster the method.

Memory complexity analysis. We study the memory (storage) requirements of a method as a function of the n DOF. These requirements depend on the storage scheme used for the matrix arising in step 2 and the solver used in step 3.

Overall efficiency analysis. Often, the most practical way of comparing two methods is to ask

1. if the methods were to run for the same length of time, which one would give the least error, or
2. given a certain error tolerance, which method satisfies that tolerance faster.

To test the overall efficiency of methods, we usually plot the error versus the time required to compute the approximate solution on a log-log scale. The method with the steepest slope is the most efficient.

11.3 Finite Difference Methods for BVPs

A FD approximation to a derivative of a function u at a point x is a linear combination of values of u at points near x (often including x). Usually, a FD approximation is first derived for some derivative of a

function of one variable, and then it is extended to partial derivatives of functions of several variables.

Let x be the point of interest and h, h_E, h_W small stepsizes. The following are several examples of FD approximations in one-dimension.

$$u_x(x) = \frac{u(x+h)-u(x)}{h} + O(h) \tag{75}$$

$$u_x(x) = \frac{u(x)-u(x-h)}{h} + O(h) \tag{76}$$

$$u_x(x) = \frac{u(x+h)-u(x-h)}{2h} + O(h^2) \tag{77}$$

$$u_x(x) = \frac{h_W^2 u(x+h_E) + (h_E^2 - h_W^2)u(x) - h_E^2 u(x-h_W)}{h_E(h_E+h_W)h_W} + O(h_E \cdot h_W) \tag{78}$$

$$u_{xx}(x) = \frac{u(x+h)-2u(x)+u(x-h)}{h^2} + O(h^2) \tag{79}$$

$$u_{xx}(x) = \frac{2h_W u(x+h_E) - 2(h_E+h_W)u(x) + 2h_E u(x-h_W)}{h_E(h_E+h_W)h_W} + O(h_E - h_W) + O([\max(h_E,h_W)]^2) \tag{80}$$

Let (x,y) be the point of interest and h, h_E, h_W, h_N, h_S small stepsizes. The following are several examples of FD approximations in two-dimensions.

$$u_{xx}(x,y) = \frac{u(x+h,y)-2u(x,y)+u(x-h,y)}{h^2} + O(h^2) \tag{81}$$

$$u_{xx}(x,y) + u_{yy}(x,y) = \frac{u(x+h,y)+u(x,y+h)-4u(x,y)+u(x,y-h)+u(x-h,y)}{h^2} + O(h^2) \tag{82}$$

$$u_{xy}(x) = \frac{u(x+h,y+h)-u(x-h,y+h)-u(x+h,y-h)+u(x-h,y-h)}{4h^2} + O(h^2) \tag{83}$$

$$u_{xy}(x) = \frac{u(x+h_E,y+h_N)-u(x-h_W,y+h_N)-u(x+h_E,y-h_S)+u(x-h_W,y-h_S)}{(h_E+h_W)(h_S+h_N)} + O(\max(h_E,h_W,h_S,h_N)) \tag{84}$$

Note the following.

- Each FD approximation listed above includes an error term. The actual FD approximation is the right side of the equation excluding the error term.
- Approximations (75)–(79) and (81)–(83) use *uniform* stepsizes, while the rest use *non-uniform* stepsizes.

- Approximations (75), (76), (80) and (84) are of *first order*, while the rest are of *second order*. The order refers to the (lowest) exponent of the stepsize(s) in the error term.
- All FD approximation formulas are derived by using appropriate Taylor series expansions around the point of approximation.
- Approximations (81)–(84) are derived by using combinations of one-dimensional Taylor series and make use of values of u at points on a *rectangular grid*.
- It is possible to derive two-dimensional FD approximations which make use of values of u at points on a *triangular*, *quadrilateral* (but not rectangular), *polygonal*, or *irregular grid* with points positioned arbitrarily. Such approximations can be derived by using two-dimensional Taylor's series.

11.3.1 An Example of a Finite Difference Method in One Dimension

Consider the problem

$$u_{xx} = g(x) \quad \text{in} \quad (0,1) \tag{85}$$

$$u = \gamma(x) \quad \text{at} \quad x = 0 \quad \text{and} \quad x = 1 \tag{86}$$

Using the FD approximation (79), we transform (85) to

$$\frac{u(x+h) - 2u(x) + u(x-h)}{h^2} = g(x) + O(h^2) \tag{87}$$

Let $\{x_i = ih : i = 0, \ldots, n\}$ with $h = 1/n$ be the set of *gridpoints* and let $U_i \approx u(x_i)$ for $i = 1, \ldots, n$. Without the $O(h^2)$ error term, the discretized PDE (87) at the gridpoint x_i becomes

$$\frac{U_{i+1} - 2U_i + U_{i-1}}{h^2} = g(x_i) \tag{88}$$

From (86), we have for the point x_1

$$\frac{U_2 - 2U_1}{h^2} = g(x_1) - \frac{\gamma(x_0)}{h^2} \tag{89}$$

By writing equation (89) first, then (88) for $i = 2, \ldots, n-2$, and finally a relation similar to (89) for the point x_{n-1}, and then multiplying each equation by h^2, we get the following linear system.

$$\begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ \vdots \\ U_{n-2} \\ U_{n-1} \end{pmatrix} = h^2 \begin{pmatrix} g(x_1) \\ g(x_2) \\ g(x_3) \\ \vdots \\ g(x_{n-2}) \\ g(x_{n-1}) \end{pmatrix} - \begin{pmatrix} \gamma(x_0) \\ 0 \\ 0 \\ \vdots \\ 0 \\ \gamma(x_n) \end{pmatrix} \tag{90}$$

Note that this system is symmetric, diagonally dominant in all rows and strictly diagonally dominant in the first and last rows. Therefore it is also positive definite and has a unique solution. By solving it, we obtain $U_i \approx u(x_i)$, $i = 1, \ldots, n-1$. Using interpolation, we can approximate u at any other point of the domain.

It can be proved that $\max\{|u(x_i) - U_i| : i = 1, \ldots, n-1\} = O(h^2)$. That is, the approximation is second order at the gridpoints.

The computational complexity of the method described is $O(n)$, since the linear system (90) is tridiagonal and of size $n-1$ (see §2.7).

11.3.2 An Example of a Finite Difference Method in Two Dimensions

Consider the problem

$$u_{xx} + u_{yy} = g(x,y) \quad \text{in} \quad (0,1) \times (0,1) \tag{91}$$

$$u = \gamma(x,y) \quad \text{on} \quad x = 0, \quad x = 1, \quad y = 0, \quad y = 1 \tag{92}$$

Using the FD approximation (82), we transform (91) to

$$\frac{u(x+h,y) + u(x,y+h) - 4u(x,y) + u(x,y-h) + u(x-h,y)}{h^2} = g(x,y) + O(h^2) \tag{93}$$

Let $\{(x_i, y_j) : x_i = ih,\ y_j = jh,\ i,j = 0, \ldots, n\}$ with $h = 1/n$ be the set of gridpoints and let $U_{ij} \approx u(x_i, y_j)$. Without the $O(h^2)$ error term, the discretized PDE (93) at the gridpoint (x_i, y_j), $i = 1, \ldots, n-1$, $j = 1, \ldots, n-1$, becomes

$$\frac{U_{i+1,j} + U_{i,j+1} - 4U_{i,j} + U_{i,j-1} + U_{i-1,j}}{h^2} = g(x_i, y_j) \tag{94}$$

From (92), we have for the point (x_1, y_1)

$$\frac{U_{1,2} + U_{2,1} - 4U_{1,1}}{h^2} = g(x_1, y_1) - \frac{\gamma(x_0, y_1) + \gamma(x_1, y_0)}{h^2} \tag{95}$$

Similar relations hold at the three other corners of the domain. Also, for the points (x_1, y_j), $j = 2, \ldots, n-2$, we have

$$\frac{U_{1,j+1} - 4U_{1,j} + U_{1,j-1} + U_{2,j}}{h^2} = g(x_1, y_j) - \frac{\gamma(x_0, y_j)}{h^2} \tag{96}$$

Similar relations hold for other gridpoints one grid line away from the boundary.

One way to number the gridpoints, and also the equations and unknowns, is bottom-up then left-to-right: $(1,1), (1,2), (1,3), \ldots, (1,n-2), (1,n-1), (2,1), (2,2), (2,3), \ldots, (2,n-2), (2,n-1), \ldots$ That is, first (95), then (96) for $j = 2, \ldots, n-2$, then relations similar to (95) for the points (x_1, y_{n-1}), (x_2, y_1), then (94) for $i = 2$, $j = 2, \ldots, n-2$, etc. Using this ordering, we get a linear system $AU = \vec{g}$, where, after multiplying

each equation by h^2, the matrix A has the form

$$\begin{pmatrix}
-4 & 1 & & & & 1 & & & & & & & & & \\
1 & -4 & 1 & & & & 1 & & & & & & & & \\
 & \ddots & \ddots & \ddots & & & & \ddots & & & & & & & \\
 & & 1 & -4 & 1 & & & & 1 & & & & & & \\
 & & & 1 & -4 & & & & & 1 & & & & & \\
1 & & & & & -4 & 1 & & & & 1 & & & & \\
 & 1 & & & & 1 & -4 & 1 & & & & 1 & & & \\
 & & \ddots & & & & \ddots & \ddots & \ddots & & & & \ddots & & \\
 & & & \ddots & & & & \ddots & \ddots & \ddots & & & & \ddots & \\
 & & & & 1 & & & & 1 & -4 & 1 & & & & 1 \\
 & & & & & 1 & & & & 1 & -4 & & & & \\
 & & & & & & 1 & & & & & -4 & 1 & & \\
 & & & & & & & 1 & & & & 1 & -4 & 1 & \\
 & & & & & & & & \ddots & & & & \ddots & \ddots & \ddots \\
 & & & & & & & & & 1 & & & & 1 & -4 & 1 \\
 & & & & & & & & & & 1 & & & & 1 & -4
\end{pmatrix}$$

This can be rewritten as

$$\begin{pmatrix}
T & I & & & \\
I & T & I & & \\
 & \ddots & \ddots & \ddots & \\
 & & I & T & I \\
 & & & I & T
\end{pmatrix}$$

where T is a tridiagonal matrix of size $n-1$ with -4's on the diagonal and 1's on the super- and sub-diagonal and I is the identity matrix of size $n-1$. Thus the matrix is *block tridiagonal* of size $(n-1)^2$. The vector of unknowns is

$$\begin{aligned}
U &= (U_{1,1}, U_{1,2}, \ldots, U_{1,n-2}, U_{1,n-1}, \\
&\quad U_{2,1}, U_{2,2}, \ldots, U_{2,n-2}, U_{2,n-1}, \ldots \\
&\quad U_{n-2,1}, U_{n-2,2}, \ldots, U_{n-2,n-2}, U_{n-2,n-1}, \\
&\quad U_{n-1,1}, U_{n-1,2}, \ldots, U_{n-1,n-2}, U_{n-1,n-1})^T
\end{aligned}$$

The right-side vector is

$$\begin{aligned}
\vec{g} &= (h^2 g_{1,1} - \gamma_{0,1} - \gamma_{1,0}, h^2 g_{1,2} - \gamma_{0,2}, \ldots, h^2 g_{1,n-2} - \gamma_{0,n-2}, h^2 g_{1,n-1} - \gamma_{0,n-1} - \gamma_{1,n}, \\
&\quad h^2 g_{2,1} - \gamma_{2,0}, h^2 g_{2,2}, \ldots, h^2 g_{2,n-2}, h^2 g_{2,n-1} - \gamma_{2,n}, \ldots,
\end{aligned}$$

$$h^2 g_{n-2,1} - \gamma_{n-2,0}, h^2 g_{n-2,2}, \ldots, h^2 g_{n-2,n-2}, h^2 g_{n-2,n-1} - \gamma_{n-2,n},$$
$$h^2 g_{n-1,1} - \gamma_{n,1} - \gamma_{n-1,0}, h^2 g_{n-1,2} - \gamma_{n,2}, \ldots,$$
$$h^2 g_{n-1,n-2} - \gamma_{n,n-2}, h^2 g_{n-1,n-1} - \gamma_{n,n-1} - \gamma_{n-1,n})^T$$

where $g_{ij} = g(x_i, y_j)$ and $\gamma_{ij} = \gamma(x_i, y_j)$. Note that this system is symmetric, diagonally dominant in all rows and strictly diagonally dominant in all rows corresponding to gridpoints one grid line away from the boundary. Therefore, it is positive definite and has a unique solution. By solving the system $AU = \vec{g}$, we obtain $U_{ij} \approx u(x_i, y_j)$ for $i = 1, \ldots, n-1$ and $j = 1, \ldots, n-1$. Using interpolation, we can approximate the value of u at any other point of the domain.

It can be proved that $\max\{|u(x_i, y_j) - U_{i,j}| : i, j = 1, \ldots, n-1\} = O(h^2)$. That is, the approximation is second order at the gridpoints.

The computational complexity of the method described above depends on the method used to solve the linear system $AU = \vec{g}$. Note that A has at most 5 non-zero entries per row, it is banded with lower and upper bandwidth $n-1$, and its size is $(n-1)^2$. If a direct band solver is used to solve $AU = \vec{g}$, then the computational complexity of the method is $O(n^4)$, but sparse direct solvers are more efficient (see §2.7). In addition, there exist iterative methods (e.g., multigrid, see §11.8 and (Briggs 1987)) which can solve this system much more efficiently, reducing the computational complexity of the method to almost $O(n^2)$.

Note that the properties of the matrix A, such as symmetry, diagonal dominance, positive-definiteness and the sparsity pattern (block tridiagonal with at most 5 non-zero entries per row), are highly dependent on the simplicity of the differential operator associated with (91) and boundary conditions (92), the choice of uniform and rectangular grid and the FD approximation (93). For a differential operator with first order derivative terms and/or Neumann BCs, symmetry is lost. Symmetry may also be lost if a non-uniform grid is chosen, even if it is rectangular. Diagonal dominance depends on the coefficients of the differential operator and on the absence of first order derivative terms. The block tridiagonal form will most likely be affected, if an irregular grid is chosen. Fast linear solvers, such as multigrid and FFT (fast Fourier transform) solvers, work well on the matrix A, but may not perform as well on more general systems. The development of fast linear solvers for such matrices is an open and active area of research. See, for example, (Van Loan 1992; Hackbusch 1994) and the references therein. For further reading on FDMs, see (Strikwerda 1989).

11.4 Finite Element Methods for BVPs

The first step in a FEM is to choose a FE approximation space and a basis for it. The most commonly used spaces are piecewise polynomials (PPs) or splines (see §8.5). Let n be the dimension of the approximation space and let $\{\phi_j(x) : j = 1, \ldots, n\}$ be a set of basis functions for the space.

Consider the problem (68)–(69). Let

$$u_\Delta(x) = \sum_{j=1}^{n} \alpha_j \phi_j(x)$$

be the approximation to u. The next step in a FEM is to choose n conditions that the approximation must satisfy. A FEM is characterized by these conditions. The most common FEMs are the *Galerkin method* and the *collocation method*.

11.4.1 The Galerkin Method

Given an inner product $(\cdot, \cdot)$, usually defined by

$$(f, g) = \int_\Omega f(x)g(x)\, dx,$$

we require that u_Δ satisfies

$$(\phi_i, Lu_\Delta - g) = 0, \qquad i = 1, \ldots, n, \tag{97}$$

forcing the *residual* $Lu_\Delta - g$ to be orthogonal to the approximation space, and making it, in a sense, as "small" as possible. If L is a linear operator, then the relations (97) are equivalent to

$$\sum_{j=1}^{n} \alpha_j(\phi_i, L\phi_j) = (\phi_i, g), \qquad i = 1, \ldots, n, \tag{98}$$

which can be written in the form $A\vec{\alpha} = \vec{g}$, where A is an $n \times n$ matrix with entries $A_{ij} = (\phi_i, L\phi_j)$, $i = 1, \ldots, n$, $j = 1, \ldots, n$, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)^T$ is the vector of coefficients and $\vec{g}$ is a vector with entries $\vec{g}_i = (\phi_i, g)$, $i = 1, \ldots, n$. We usually use numerical integration to compute the entries of A and $\vec{g}$ (see §9).

As an example of the Galerkin method in one dimension, consider the problem (85)–(86) and the set of gridpoints $\{x_i = ih : i = 0, \ldots, n\}$ with $h = 1/n$. Let $\{\phi_i : i = 0, \ldots, n\}$ be the set of linear spline basis functions w.r.t. the knots (gridpoints) $\{x_i\}$, as defined by (39). Then

$$u_\Delta(x) = \sum_{i=0}^{n} \alpha_i \phi_i(x)$$

is the linear spline approximation to u. From the BC (86), we get $u_\Delta(x_0) = \gamma(x_0)$, which implies $\alpha_0 = \gamma(x_0)$. Similarly, $\alpha_n = \gamma(x_n)$. The remaining unknowns $\{\alpha_i : i = 1, \ldots, n-1\}$ are determined by the $n-1$ Galerkin conditions

$$\sum_{j=0}^{n} \alpha_j(\phi_i, L\phi_j) = (\phi_i, g), \qquad i = 1, \ldots, n-1$$

which, for the particular L associated with (85), are equivalent to

$$\sum_{j=0}^{n} \alpha_j(\phi_i, \phi_j'') = (\phi_i, g), \qquad i = 1, \ldots, n-1.$$

Writing the inner product (ϕ_i, ϕ_j'') as an integral and applying integration by parts, these conditions reduce

to

$$\sum_{j=1}^{n-1} \alpha_j \int_0^1 \phi_i' \phi_j' \, dx = \left[\phi_i \sum_{j=0}^{n} \alpha_j \phi_j' \right]_0^1 - \int_0^1 \phi_i g \, dx - \alpha_0 \int_0^1 \phi_i' \phi_0' \, dx - \alpha_n \int_0^1 \phi_i' \phi_n' \, dx, \qquad (99)$$
$$i = 1, \ldots, n-1.$$

Note that the term

$$\left[\phi_i \sum_{j=0}^{n} \alpha_j \phi_j' \right]_0^1 = 0,$$

since $\phi_i(0) = \phi_i(1) = 0$ for $i = 1, \ldots, n-1$. Relations (99) form a linear system of size $n-1$; the associated matrix A has elements

$$A_{i,j} = \int_0^1 \phi_i' \phi_j' \, dx.$$

Since the basis functions $\{\phi_i\}$ are non-zero on at most 2 subintervals, A is tridiagonal with elements

$$\begin{aligned} A_{i,i} &= \int_{x_{i-1}}^{x_{i+1}} \phi_i' \phi_i' \, dx, \qquad i = 1, \ldots, n-1, \\ A_{i,i-1} &= \int_{x_{i-1}}^{x_i} \phi_i' \phi_{i-1}' \, dx, \qquad i = 2, \ldots, n-1, \\ A_{i,i+1} &= \int_{x_i}^{x_{i+1}} \phi_i' \phi_{i+1}' \, dx, \qquad i = 1, \ldots, n-2. \end{aligned}$$

It can be proved that this matrix is also symmetric positive-definite. Thus, the associated system has a unique solution. By solving the system, we obtain the coefficients $\{\alpha_i : i = 0, \ldots, n\}$ of u_Δ, which we can evaluate at any point of the domain $(0, 1)$.

It can be proved that $\max\{|u(x) - u_\Delta(x)| : x \in [0, 1]\} = O(h^2)$. That is, the approximation is second order on the whole domain.

The computational complexity of the method described is $O(n)$, since the linear system that has to be solved is tridiagonal and of size $n-1$ (see §2.7).

Relations (99) can also be derived using a variational method (see §11.1.1). Thus, for problem (85)–(86), there is a variational method equivalent to the Galerkin method. This is true for all differential equation problems with a self-adjoint positive-definite operator. There exist differential equation problems, though, which are not characterized by variational principles. In such cases, the Galerkin method is applicable, while the variational method is not.

As an example in two dimensions, consider problem (91)–(92) with the gridpoints $\{(x_i, y_j) : x_i = ih, \; y_j = jh, \; i, j = 0, \ldots, n\}$ for $h = 1/n$. A common way to define an approximation space for two-dimensional problems is to choose a tensor product of approximation spaces in each dimension. Let $\{\phi_i(x) : i = 0, \ldots, n\}$ be the linear spline basis functions w.r.t. the knots $\{x_i : i = 0, \ldots, n\}$ and let $\{\phi_j(y) : j = 0, \ldots, n\}$ be the

linear spline basis functions w.r.t. the knots $\{y_j : j = 0, \ldots, n\}$, as defined in (39). Then

$$u_\Delta(x, y) = \sum_{i=0}^{n} \sum_{j=0}^{n} \alpha_{ij} \phi_i(x) \phi_j(y)$$

is the bilinear spline approximation to u. Continuing as in the one-dimensional case, we derive a system of $(n+1)^2$ equations in $(n+1)^2$ unknowns. The associated matrix A is block-tridiagonal, with at most 9 non-zero entries per row and bandwidth $n+2$. It is also symmetric positive-definite. Thus, the associated system has a unique solution. Moreover, it can be proved that the approximation u_Δ is second order.

Note that, if instead of a rectangular subdivision of the domain and bilinear elements, we choose a triangular subdivision and linear elements (w.r.t x and y), we would get a system similar to that of §11.3.2.

An important property of the Galerkin method is that, for any self-adjoint positive-definite differential operator, the resulting matrix is symmetric positive-definite, even if the grid is irregular. As stated before, for all differential equation problems with a self-adjoint and positive-definite operator, there is a variational method equivalent to the Galerkin method. This holds for higher dimension problems too. Thus, large, sparse, symmetric, positive-definite matrices arise from the application of variational methods.

For an introduction to the FEM, including its computer implementation, see (Becker, Carey and Oden 1981). An error analysis is carried out in (Strang and Fix 1973).

11.4.2 The Collocation Method

We first pick n *collocation points* $\{t_i : i = 1, \ldots, n\}$ in Ω and on $\partial\Omega$. We then require that u_Δ satisfies

$$Lu_\Delta(t_i) - g(t_i) = 0, \quad \text{if } t_i \in \Omega \tag{100}$$

$$Bu_\Delta(t_i) - \gamma(t_i) = 0, \quad \text{if } t_i \in \partial\Omega \tag{101}$$

forcing the residuals $Lu_\Delta - g$ and $Bu_\Delta - \gamma$ to be zero at the collocation points, and making them, in a sense, as "small" as possible. If L and B are linear, relations (100)–(101) are equivalent to

$$\sum_{j=1}^{n} \alpha_j L\phi_j(t_i) = g(t_i), \quad \text{if } t_i \in \Omega \tag{102}$$

$$\sum_{j=1}^{n} \alpha_j B\phi_j(t_i) = \gamma(t_i), \quad \text{if } t_i \in \partial\Omega \tag{103}$$

which can be written in the form $A\vec{\alpha} = \vec{g}$, where A is an $n \times n$ matrix with entries $A_{ij} = L\phi_j(t_i)$, $j = 1, \ldots, n$, for all $t_i \in \Omega$ and $A_{ij} = B\phi_j(t_i)$, $j = 1, \ldots, n$, for all $t_i \in \partial\Omega$, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)^T$ is the vector of coefficients and $\vec{g}$ is a vector with entries $\vec{g}_i = g(t_i)$ for all $t_i \in \Omega$ and $\vec{g}_i = \gamma(t_i)$ for all $t_i \in \partial\Omega$.

The choice of collocation points is critical to the success of the method. It affects not only the solvability and other properties (such as symmetry, diagonal dominance, bandedness) of the matrix A but also the accuracy of the approximation u_Δ. Depending on the FE approximation space which u_Δ belongs to, some standard choices of collocation points in one dimension are listed below.

- If the FE approximation space is the space of quadratic splines (quadratic PPs in $\mathcal{C}^1$), the collocation points are chosen to be the midpoints of the subintervals (x_{i-1}, x_i), $i = 1, \ldots, n$ and the two boundary points. The same choice of collocation points is effective if the FE approximation space is composed of any other even degree splines, with the exception that some additional collocation conditions may be required at boundary points or points close to the boundary.

- If the FE approximation space is the space of cubic splines (cubic PPs in $\mathcal{C}^2$), the collocation points are chosen to be the gridpoints $\{x_i : i = 0, \ldots, n\}$. At each of the boundary points, x_0 and x_n, both conditions (100) and (101) are imposed. The same choice of collocation points is effective if the FE approximation space is composed of any other odd degree splines, with the exception that some additional collocation conditions may be required at boundary points or points close to the boundary.

- If the FE approximation space is the space of cubic PPs in $\mathcal{C}^1$ (cubic Hermite PPs), the collocation points are chosen to be the two Gauss points $x_{i-1} + (3 \pm \sqrt{3})(x_i - x_{i-1})/6$ in each subinterval (x_{i-1}, x_i), $i = 1, \ldots, n$, and the two boundary gridpoints.

As an example in one dimension, consider problem (85)–(86) and the set of gridpoints $\{x_i = ih : i = 0, \ldots, n\}$ with $h = 1/n$. Let the collocation points be the midpoints $t_i = (x_{i-1} + x_i)/2$, $i = 1, \ldots, n$, and the end-points $t_0 = x_0$ and $t_{n+1} = x_n$. Let $\{\phi_i : i = 0, \ldots, n+1\}$ be the quadratic spline basis functions w.r.t. the knots (gridpoints) $\{x_i\}$, as defined in (40). Then

$$u_\Delta(x) = \sum_{i=0}^{n+1} \alpha_i \phi_i(x)$$

is the quadratic spline approximation to u. Relation (100) for the PDE (85) becomes $u''_\Delta(t_i) = g(t_i)$, so relation (102) becomes

$$\alpha_{i-1}\phi''_{i-1}(t_i) + \alpha_i \phi''_i(t_i) + \alpha_{i+1}\phi''_{i+1}(t_i) = g(t_i)$$

which reduces to

$$\frac{\alpha_{i-1} - 2\alpha_i + \alpha_{i+1}}{h^2} = g(t_i), \qquad i = 1, \ldots, n. \tag{104}$$

Relation (101) for the BC (86) becomes $u_\Delta(t_0) = \gamma(t_0)$, so relation (103) becomes

$$\alpha_0 \phi_0(t_0) + \alpha_1 \phi_1(t_0) = \gamma(t_0)$$

which reduces to

$$\frac{\alpha_0 + \alpha_1}{2} = \gamma(t_0) \tag{105}$$

Similarly, the collocation condition at $t_{n+1} = 1$ reduces to

$$\frac{\alpha_n + \alpha_{n+1}}{2} = \gamma(t_{n+1}) \tag{106}$$

Writing (105) first, then (104) for $i = 1, \ldots, n$ and finally (106), we get a tridiagonal system of equations

w.r.t. the coefficients $\{\alpha_i : i = 0, \ldots, n+1\}$. The system is diagonally dominant and it can be proved that it has a unique solution. It can also be scaled so that it is symmetric positive-definite.

It can be proved that $\max\{|u(x) - u_\Delta(x)| : x \in [0,1]\} = O(h^2)$. That is, the approximation is second order on the whole domain. There exists a variant of this method, though, which is fourth order at the gridpoints and midpoints and third order on the whole domain (Houstis, Christara and Rice 1988).

The computational complexity of the method described above is $O(n)$, since the linear system that must be solved is tridiagonal and of size $n+1$ (see §2.7).

As an example in two dimensions, consider problem (91)–(92) and the set of gridpoints, $\{(x_i, y_j) : x_i = ih,\ y_j = jh,\ i, j = 0, \ldots, n\}$ with $h = 1/n$. A common approximation space for two-dimensional problems is a tensor product of approximation spaces in each dimension. Let $\{\phi_i(x) : i = 0, \ldots, n+1\}$ be the quadratic spline basis functions w.r.t. the knots (gridpoints) $\{x_i : i = 0, \ldots, n\}$ and let $\{\phi_j(y) : j = 0, \ldots, n+1\}$ be the quadratic spline basis functions w.r.t. the knots $\{y_j : j = 0, \ldots, n\}$, as defined in (40). Then

$$u_\Delta(x, y) = \sum_{i=0}^{n+1} \sum_{j=0}^{n+1} \alpha_{ij} \phi_i(x) \phi_j(y)$$

is the bi-quadratic spline approximation to u. Continuing as in the one-dimensional case, we derive a system of $(n+2)^2$ equations and unknowns. The associated matrix is block-tridiagonal, with at most 9 non-zero entries per row, and has bandwidth $n+3$. It can be proved that this system has a unique solution and that the approximation u_Δ is second order. With appropriate modifications, though, the order can be improved as in the one-dimensional case (Christara 1994).

For a general introduction to collocation methods, see (Prenter 1975).

11.5 Finite Difference Methods for IVPs

Consider the problem (70)–(72). Let the temporal gridpoints be $\{t_j = jh_t : j = 0, \ldots, m\}$ with $h_t = T/m$. Starting with the initial values of u at t_0, given by (72), most FDMs for IVPs compute approximate values of u at each subsequent temporal gridpoint t_j, in the order $j = 1, \ldots, m$, using previous and/or current approximate values of u at neighbouring space points.

If, at each temporal gridpoint t_j, a method uses only approximations from previous temporal gridpoints, it is called *explicit*, as it does not require the solution of a system of equations to proceed from one temporal gridpoint to the next. If, at some temporal gridpoint t_j, a method uses approximations from the current temporal gridpoint t_j, it is called *implicit*, as it requires the solution of a system of equations to proceed from one temporal gridpoint to the next. If, at the time step from t_{j-1} to t_j, a method uses approximations from t_{j-1} and t_j only, it is called *one-step*. Likewise, we can define *two-step* methods, etc. These definitions for PDEs are similar to those given in §10 for ODEs.

11.5.1 An Example of an Explicit One-Step Method for a Parabolic IVP

Consider the problem

$$u_t = u_{xx} \quad \text{in} \quad (0,1) \times (0,T) \tag{107}$$

$$u = \gamma_0(t) \quad \text{on} \quad x = 0,\ t \in (0,T) \tag{108}$$

$$u = \gamma_1(t) \quad \text{on} \quad x = 1,\ t \in (0,T) \tag{109}$$

$$u = g(x) \quad \text{on} \quad t = 0,\ x \in [0,1] \tag{110}$$

Using the FD approximations (79) for u_{xx} and (75) for u_t, we transform (107) to

$$\frac{u(x,t+h_t) - u(x,t)}{h_t} = \frac{u(x+h,t) - 2u(x,t) + u(x-h,t)}{h^2} + O(h_t + h^2) \tag{111}$$

Let $\{x_i = ih : i = 0, \ldots, n\}$ with $h = 1/n$ be the set of spatial gridpoints and $\{t_j = jh_t : j = 0, \ldots, m\}$ with $h_t = T/m$ be the set of temporal gridpoints. Also let $U_{i,j} \approx u(x_i, t_j)$ for $i = 0, \ldots, n$ and $j = 0, \ldots, m$. Then the discretized PDE (111) at the point (x_i, t_j), $i = 1, \ldots, n-1$, $j = 1, \ldots, m$, becomes

$$\frac{U_{i,j+1} - U_{i,j}}{h_t} = \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{h^2}$$

Letting $r = h_t/h^2$, we can rewrite this relation as

$$U_{i,j+1} = rU_{i+1,j} + (1-2r)U_{i,j} + rU_{i-1,j} \tag{112}$$

for $i = 2, \ldots, n-2$. For $i = 1$, we have from (108)

$$U_{1,j+1} = rU_{2,j} + (1-2r)U_{1,j} + r\gamma_0(t_j) \tag{113}$$

Similarly, for $i = n-1$, we have from (109)

$$U_{n-1,j+1} = r\gamma_1(t_j) + (1-2r)U_{n-1,j} + rU_{n-2,j} \tag{114}$$

For $j = 1$, we have from (110)

$$U_{i,1} = rg(x_{i+1}) + (1-2r)g(x_i) + rg(x_{i-1}) \tag{115}$$

Thus, we can compute $U_{i,j} \approx u(x_i, t_j)$ from a linear combination of three neighbouring spatial approximations at t_{j-1}.

It can be proved that, if $r < 1/2$, then $\max\{|u(x_i,t_j) - U_{i,j}| : i = 1, \ldots, n,\ j = 1, \ldots, m\} = O(h^2 + h_t)$, thus the order of convergence is one w.r.t. to h_t and two w.r.t. h. It can also be proved that, if $r < 1/2$, then the method is stable. However, the restriction $r < 1/2$ may be impractical for many problems, since it forces h_t to be very small if h is small and so the method must take many steps to integrate the problem.

The computational complexity of the method is $O(nm)$, since for each gridpoint (x_i, t_j) a constant number of floating-point operations must be performed.

11.5.2 An Example of an Implicit One-Step Method for a Parabolic IVP

Consider the problem (107)–(110) once more. Using the FD approximations (79) for u_{xx} and (76) for u_t, we transform (107) to

$$\frac{u(x,t+h_t)-u(x,t)}{h_t} = \frac{u(x+h,t+h_t)-2u(x,t+h_t)+u(x-h,t+h_t)}{h^2} + O(h_t+h^2) \tag{116}$$

Again let $\{x_i = ih : i = 0,\ldots,n\}$ with $h = 1/n$ be the set of spatial gridpoints and $\{t_j = jh_t : j = 0,\ldots,m\}$ with $h_t = T/m$ be the set of temporal gridpoints. Also let $U_{i,j} \approx u(x_i,t_j)$ for $i = 0,\ldots,n$ and $j = 0,\ldots,m$. Then, the discretized PDE (116) at the point (x_i,t_j), $i = 1,\ldots,n-1$, $j = 1,\ldots,m$, becomes

$$\frac{U_{i,j+1}-U_{i,j}}{h_t} = \frac{U_{i+1,j+1}-2U_{i,j+1}+U_{i-1,j+1}}{h^2}$$

Letting $r = h_t/h^2$ again, we can rewrite this relation as

$$-rU_{i-1,j+1} + (1+2r)U_{i,j+1} - rU_{i+1,j+1} = U_{i,j} \tag{117}$$

for $i = 2,\ldots,n-2$. For $i = 1$, we have from (108)

$$(1+2r)U_{1,j+1} - rU_{2,j+1} = U_{1,j} + r\gamma_0(t_{j+1}) \tag{118}$$

Similarly, for $i = n-1$, we have from (109)

$$-rU_{n-2,j+1} + (1+2r)U_{n-1,j+1} = U_{n-1,j} + r\gamma_1(t_{j+1}) \tag{119}$$

For $j = 1$, we have from (110)

$$-rU_{i-1,1} + (1+2r)U_{i,1} - rU_{i+1,1} = g(x_i) \tag{120}$$

Thus, at the j-th time step, a tridiagonal linear system must be solved to compute $U_{i,j} \approx u(x_i,t_j)$. The diagonal entries of the associated matrix are all equal to $(1+2r)$, while the off-diagonal entries are all equal to $-r$. The system is symmetric positive-definite and strictly diagonally dominant, thus it has a unique solution.

It can be proved that $\max\{|u(x_i,t_j) - U_{i,j}| : i = 1,\ldots,n,\ j = 1,\ldots,m\} = O(h^2+h_t)$, thus the order of convergence is one w.r.t. to h_t and two w.r.t. h. It can also be proved that the method is stable without any restrictions on r (except $r > 0$).

The computational complexity of the method is $O(nm)$, since at each time step we must solve a tridiagonal linear system of size $n-1$ (see §2.7).

Note that, for the problem (107)–(110), which is one-dimensional w.r.t. to space, both the explicit and implicit methods have the same computational complexity. This is not true for problems in more space dimensions. For such problems, the solution of a linear system at each time step can be very time consuming, making an implicit method much more expensive per step than an explicit one. However, because there is no restriction on r for some implicit schemes, while there always is for an explicit one, some implicit schemes may be able to take far fewer timesteps than an implicit one. As a result, an implicit method may be computationally more efficient than an explicit one.

11.5.3 An Example of an Explicit Two-Step Method for a Hyperbolic IVP

Consider the problem

$$\begin{aligned} u_{tt} &= u_{xx} \quad \text{in} \quad (0,1)\times(0,T) && (121)\\ u &= \gamma_0(t) \quad \text{on} \quad x=0,\ t\in(0,T) && (122)\\ u &= \gamma_1(t) \quad \text{on} \quad x=1,\ t\in(0,T) && (123)\\ u &= g_0(x) \quad \text{on} \quad t=0,\ x\in[0,1] && (124)\\ u_t &= g_1(x) \quad \text{on} \quad t=0,\ x\in[0,1] && (125) \end{aligned}$$

Using the FD approximation (79) for u_{xx} and u_{tt}, we transform (121) to

$$\frac{u(x,t+h_t)-2u(x,t)+u(x,t-h_t)}{h_t^2} = \frac{u(x+h,t)-2u(x,t)+u(x-h,t)}{h^2} + O(h_t^2+h^2) \tag{126}$$

Again let $\{x_i = ih : i=0,\ldots,n\}$ with $h=1/n$ be the set of spatial gridpoints and $\{t_j = jh_t : j=0,\ldots,m\}$ with $h_t = T/m$ be the set of temporal gridpoints. Also let $U_{i,j} \approx u(x_i,t_j)$ for $i=0,\ldots,n$ and $j=0,\ldots,m$. Then, the discretized PDE (126) at the point (x_i,t_j), $i=1,\ldots,n-1$, $j=1,\ldots,m$, becomes

$$\frac{U_{i,j+1}-2U_{i,j}+U_{i,j-1}}{h_t^2} = \frac{U_{i+1,j}-2U_{i,j}+U_{i-1,j}}{h^2}$$

Letting $r = h_t/h$, we can rewrite this relation as

$$U_{i,j+1} = r^2U_{i-1,j} + 2(1-r^2)U_{i,j} + r^2U_{i+1,j} - U_{i,j-1} \tag{127}$$

for $i=2,\ldots,n-1$. For gridpoints close to the boundary, the approximate values of U are replaced by the values of the functions γ_0 and γ_1 at the appropriate points, as in §11.5.1 and §11.5.2. Thus, we can compute $U_{i,j+1} \approx u(x_i,t_{j+1})$ from a linear combination of three neighbouring spatial approximations at time t_j and one approximation at time t_{j-1}.

Since (127) is a two-step formula, at the initial time-point t_0 it cannot be applied as is. At that point, we use the ICs (124)–(125) and the FD approximation (75) to get

$$U_{i,1} = g_0(x_i) + h_t g_1(x_i) \tag{128}$$

It can be proved that, if $r < 1$, then the method is stable. It can also be shown that $\max\{|u(x_i, t_j) - U_{i,j}| : i = 1, \ldots, n,\ j = 1, \ldots, m\} = O(h^2 + h_t^2)$, thus the order of convergence is two w.r.t. to both h_t and h. Note that the restriction $r < 1$ is not impractical in this case, since it requires only that $h_t < h$.

The computational complexity of the method is $O(nm)$, since for each gridpoint (x_i, t_j) we apply a formula with a constant number of floating-point operations.

11.6 The Method of Lines

The general idea behind the *method of lines* (MOL) is to use an ODE solver along one of the dimensions of the PDE, while using a PDE discretization across the other dimensions. In its most common form for the solution of IVPs for PDEs, an ODE solver is used along the temporal dimension, while a PDE discretization is employed across the spatial dimensions, transforming an IVP for a PDE into a system of IVPs for ODEs.

To see how this is done, consider the problem (107)–(110) again. Let

$$u_\Delta(x,t) = \sum_{j=1}^{n} \alpha_j(t)\phi_j(x)$$

be a FE approximation to the true solution $u(x,t)$. Now apply a FEM condition to u_Δ to discretize the PDE (107) w.r.t. the spatial dimension. For example, collocation at the points x_i, $i = 1, \ldots, n$, yields

$$\sum_{j=1}^{n} \alpha_j'(t)\phi_j(x_i) = \sum_{j=1}^{n} \alpha_j(t)\phi_j''(x_i)$$

Let $\vec{\alpha}(t) = (\alpha_1(t), \ldots, \alpha_n(t))^T$, Φ be the matrix with entries $\Phi_{ij} = \phi_j(x_i)$, and A be the matrix with entries $A_{ij} = \phi_j''(x_i)$. Then the PDE (107) is approximated by the system of ODEs $\Phi\vec{\alpha}'(t) = A\vec{\alpha}(t)$.

To obtain an IC for the ODE, we construct an interpolant g_Δ of g in the same space as that spanned by $\{\phi_j : j = 1, \ldots, n\}$. Let

$$g_\Delta(x) = \sum_{j=1}^{n} \beta_i\phi_i(x)$$

be the FE representation of g_Δ in that space and set $\vec{\beta} = (\beta_1, \ldots, \beta_n)^T$. Then

$$\Phi\vec{\alpha}'(t) = A\vec{\alpha}(t) \tag{129}$$

$$\vec{\alpha}(0) = \vec{\beta} \tag{130}$$

is a well defined IVP for ODEs. Thus, the PDE problem (107)–(110) is converted to an IVP for a system of n ODEs. The latter can be solved by the techniques described in §10.

Note that applying an ODE method to discretize the IVP (129)–(130) results in a discretization for the PDE (107)–(110). That is, the MOL produces a discretization for a PDE. However, it is generally agreed that standard software for ODEs is more highly developed than for PDEs. Thus, using the MOL to decouple the discretization of the spatial and temporal variables allows us to exploit easily sophisticated time-stepping

techniques. As a result, the MOL is often the simplest effective method to solve a PDE.

11.7 Boundary Element Methods

The general idea behind *boundary element methods* (BEMs) is to transform the PDE to an integral equation in which the integrations take place along the boundary only of the PDE domain, thus eliminating the need for domain discretization and reducing the dimension of the PDE by one. For example, a one-dimensional integral equation is solved instead of an equivalent two-dimensional PDE. The BEM is applicable to BVPs for Laplace's or Poisson's equation, and many other simple PDEs. If applicable, this approach is often very effective, especially when the PDE domain is highly irregular.

11.8 The Multigrid Method

The multigrid method (MM) exploits the connection between a physical problem and its matrix analogue to accelerate the convergence of an iterative method (see §3). For simplicity, we describe the MM for the one-dimensional problem (85)–(86), although the merits of the scheme become apparent for two- and higher-dimensional problems (see §11.3.2). Also, we illustrate the technique using Jacobi's method as the basic iterative scheme. The MM, though, can be used with many other iterative methods as a preconditioning technique.

Let A be the matrix in (90). Apply Jacobi's method (see §3.1) with an extra damping-factor of 2 to the linear system (90). The associated iteration matrix is $G = I - A/4$. It can be shown that the eigenvalues of G are $\mu_i = \cos^2(i\pi/(2n))$, $i = 1, \ldots, n-1$, and that the components of the eigenvector v_i associated with μ_i are $\sin(i\pi(j/n))$, $j = 1, \ldots, n-1$. These are also the eigenvectors of A. Since $\{v_i : i = 1, \ldots, n-1\}$ spans $\mathbb{R}^{n-1}$, we can write the error e_0 associated with the initial guess for the damped Jacobi iteration as

$$e_0 = \sum_{i=1}^{n-1} \alpha_i v_i$$

for some scalars $\{\alpha_i : i = 1, \ldots, n-1\}$. It then follows easily from the discussion in §3.1 that the error at iteration k is

$$e_k = \sum_{i=1}^{n-1} \alpha_i \mu_i^k v_i.$$

The terms of the sum corresponding to small values of i are called *low-frequency components*, while those corresponding to large values of i are called *high-frequency components*. Note that $0 < \mu_{n-1} < \mu_{n-2} < \cdots < \mu_2 < \mu_1 < 1$. Moreover, $\mu_1 \approx 1 - (\pi/(2n))^2$, while $\mu_{n-1} \approx (\pi/(2n))^2$. Consequently, $e_k \to 0$ as $k \to \infty$, but the low-frequency components of the error converge slowly, while the high-frequency components converge rapidly.

To accelerate the convergence of the low frequency components of the error, consider solving the problem (90) on a coarse grid with $\hat{n} = n/2$ subintervals and $\hat{n} + 1$ gridpoints, assuming for simplicity that n is even. Although the coarse grid has about half the gridpoints of the fine one, the $\hat{n} - 1$ eigenvectors of the matrix

$\hat{A}$ for the coarse grid provide a good representation of the low to middle frequency eigenvectors of A. As a result, the solution to the problem (90) on the coarse grid provides a good approximation to the low to middle frequency components of the fine-grid solution. This suggests that the coarse-grid solution can be used to provide good approximations to the low to middle frequency components of the fine-grid solution, while the damped Jacobi iteration on the fine grid can be used to provide good approximations to the middle to high frequency components of the fine-grid solution.

This is the motivation behind the MM. The term "multigrid" refers to the use of several levels of grids (possibly a fine grid, several intermediate level grids and a coarse grid), so that each level damps certain components of the error fast.

To view the MM as a preconditioning technique, consider the linear system $Au = g$ corresponding to the discretization of problem (85)–(86) on some fine grid. Apply one (or a few) damped Jacobi iteration(s) to $Au = g$ to obtain an approximate solution vector $\tilde{u}$. Let $r = g - A\tilde{u}$ be the *residual* vector. Project r to a coarse grid to obtain the *coarse-grid residual* vector $\hat{r}$. This can be done by appropriately interpolating the components of r and evaluating the interpolant at the points of the coarse grid (see §8). Let $\hat{A}$ be the matrix corresponding to the discretization of problem (85)–(86) on the coarse grid. Solve (or approximately solve) $\hat{A}\tilde{r} = \hat{r}$. This can be done by applying a few damped Jacobi iterations, or by recursively applying the MM to $\hat{A}\tilde{r} = \hat{r}$, or by using a direct solver (see §2), since $\hat{A}$ is a smaller matrix than A. Now $\tilde{r}$ is the *preconditioned coarse-grid residual* vector. Extend $\tilde{r}$ to the fine grid to obtain the *preconditioned fine-grid residual* vector $\bar{r}$. This can be done by appropriately interpolating the components of $\tilde{r}$ and evaluating the interpolant at the points of the fine grid. Add $\bar{r}$ to $\tilde{u}$ to obtain a new approximate solution vector. Repeat the process until convergence. Usually, only a few iterations are needed. Note that this scheme has some similarities to iterative improvement, as described in §2.9.

The power of the MM lies in the fact that the coarse grid, which acts as a preconditioner, allows the information to pass from a point of the problem domain to another point in a few steps, while the fine grid maintains the accuracy required. Note that the interpolation and the evaluation of the interpolant needed for the projection of a fine-grid vector to a coarse-grid vector and for the extension of a coarse-grid vector to a fine-grid vector often reduce to simple relations, such as averaging neighbouring vector components. A practical introduction to the MM, including an error analysis, can be found in (Briggs 1987).

12 Parallel Computation

The increasing demand by scientists and engineers to solve larger and larger problems constitutes the primary motivation for parallel computation. Another impetus is the cost effectiveness of computers consisting of many standard CPUs compared to those based on one very fast CPU.

A parallel computer has the ability to execute simultaneously many different processes by having several independent processors concurrently perform operations on different data. A parallel computer in which all processors perform the same operation on different data is called a *Single-Instruction-Multiple-Data* (SIMD) machine, while one in which each processor has the ability to perform different operation(s) on data is called

a *Multiple-Instruction-Multiple-Data* (MIMD) machine. Sometimes a processor may use data computed by another processor, either by exchanging messages with it or by sharing some common memory area. The former is characteristic of *distributed memory* or *message passing* machines, while the latter is characteristic of *shared memory* machines.

A vector computer may be viewed as a restricted form of a parallel machine. On such a computer, one arithmetic operation is performed by several (usually a few) processors, which cooperate in a pipelined manner. Each processor receives an input operand from another processor, performs a part of the operation on it, and passes the result on to the next processor. While a processor performs its part of the operation on an operand, the next processor performs its part of the operation on the previous operand, etc. This pipelining technique is very effective when the same operation is performed on many data — for example, on each component of a long vector.

To utilize parallel computers effectively, we must be able to split the computation into *parallel processes* to be assigned to different processors. To this end, it is desirable to have fully independent computations in each process. For many problems, though, this is impossible, but there are techniques to minimize the dependence of the computation of one process on another. It is also desirable to have almost equal amounts of computation in each process for *load-balancing*. This is also hard to accomplish in many cases, but again there exist techniques to achieve reasonable load-balancing.

Many computational scientists study the parallelization of existing numerical methods, as well as the development of new methods appropriate for parallel machines. In the next subsection, we discuss, as a simple example, a numerical method which is effective for solving tridiagonal linear systems on parallel computers.

12.1 Cyclic Reduction

Consider solving the tridiagonal system

$$\begin{pmatrix} a_1 & c_1 & & & & \\ b_2 & a_2 & c_2 & & & \\ & b_3 & a_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & b_{n-1} & a_{n-1} & c_{n-1} \\ & & & & b_n & a_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}$$

by the following LU factorization algorithm for tridiagonal matrices.

for $k = 1, \ldots, n-1$ do
    $b_{k+1} = b_{k+1}/a_k$
    $a_{k+1} = a_{k+1} - b_{k+1}c_k$
end

At the end of the computation, the modified b's form the sub-diagonal of the unit lower triangular matrix L and the modified a's and c's form the diagonal and super-diagonal, respectively, of the upper triangular matrix U (see §2.3 and §2.7).

Note that the computation proceeds in the order $b_2, a_2, b_3, a_3, \ldots$ Each value computed depends on the previous one. Therefore, it seems that the computation is purely sequential and that there is no easy way to parallelize it. However, there are other ways to solve tridiagonal linear systems, and some of them can be implemented effectively on a parallel machine.

Assume, for simplicity, that $n = 2^q - 1$, where q is a positive integer. Multiply row 1 by b_2/a_1 and subtract it from row 2, eliminating x_1 from row 2. Also multiply row 3 by c_2/a_3 and subtract it from row 2, eliminating x_3 from row 2. The new row 2 involves variables x_2 and x_4 only.

Repeat the process described above for the $(n-1)/2$ groups of rows $(3, 4, 5)$, $(5, 6, 7)$, etc. This eliminates the odd unknowns from the even equations. The even equations form a new tridiagonal linear system of about half the size, $(n-1)/2 = 2^{q-1} - 1$, called the *reduced* system. This technique is often called *odd-even reduction*.

Now apply odd-even reduction to the reduced system to obtain another reduced system that is again about half as big as the first reduced system. The recursive application of odd-even reduction continues for $q = \log_2(n+1)$ steps. At each step, the even equations of the previous step form a reduced tridiagonal system of about half the size of the system from the previous step. At the end of step q, one equation in one unknown remains, so that unknown can be computed easily. This recursive technique is often called *cyclic reduction*.

Then the computation continues in the reverse order with a process called *back substitution*. At each step of back substitution, the even variables are known from the solution of the associated reduced system of about half the size. Substituting these values back into the odd equations of the larger system, we can easily compute all the odd variables.

Both the cyclic reduction algorithm and the back substitution algorithm require $q = \log_2(n+1)$ steps each. In cyclic reduction, the number of floating-point operations is divided by 2 at each step, starting with $O(n)$ floating-point operations in the first step. Thus, it requires $O(n \log n)$ arithmetic operations. Similarly, back substitution also requires $O(n \log n)$ arithmetic operations. Therefore, the computational complexity of the full solve is $O(n \log n)$.

Observe that the algorithm described above is highly parallel. The elimination operations applied to a group of three rows to obtain the reduced system at each step are independent of the elimination operations applied to any other group of three rows, and so can be carried out in parallel. Similarly, the substitution operations to compute the odd unknowns in a reduced system given the even ones are independent of each other. Thus, the unknowns of each back substitution step can also be computed in parallel.

Assume that we have $p = (n-1)/2$ processors. Initially, processor 1 is assigned rows $(1, 2, 3)$, processor 2 is assigned rows $(3, 4, 5)$, processor 3 is assigned rows $(5, 6, 7)$, and so on. After the first odd-even reduction step, processor 2 will use equation 2 from processor 1, equation 4 from itself and equation 6 from processor 3. Similarly, processor 4 will use equation 6 from processor 3, equation 8 from itself and equation 10 from

processor 5, and so on. Only the even processors will continue. The procedure is repeated. The final reduced system is solved by one processor. For the back substitution, 1 processor works first, then 2, then 4, and so on.

Thus, the algorithm requires $2\log_2(n-1)$ steps with a constant amount of computation done on each processor per step, So the parallel computational complexity of the algorithm is $O(\log n)$, which is a factor of $O(n) = O(p)$ improvement over the $O(n \log n)$ computational complexity of the serial version of the algorithm, and a little less than $O(n)$ improvement over the $O(n)$ serial computational complexity of the standard LU factorization algorithm for tridiagonal systems. So we can say that, asymptotically, the algorithm has perfect *speedup*.

Note that, when a processor uses rows computed by another processor, some communication and/or synchronization must take place between processors. This may degrade the parallel performance of the algorithm from the perfect asymptotic performance. The time spent in communication and/or synchronization depends heavily on the way the processors cooperate. More specifically, it depends on the interconnection network between processors and on the implementation of specific hardware instructions.

For an introduction to parallel numerical methods, see (Bertsekas and Tsitsiklis 1989; Ortega 1988; Van de Velde 1994).

13 Sources of Numerical Software

Although most of this article has dealt with elementary numerical methods, we strongly recommend that readers do not program these schemes themselves. High-quality software incorporating these — or more sophisticated numerical methods — is readily available. In addition, good library routines often contain many additional strategies and heuristics (not discussed here) to improve their efficiency and reliability. Using such routines, rather than attempting to re-program them, will likely save readers a significant amount of time as well as produce superior numerical results.

We highly recommend that readers familiarize themselves with the *Guide to Available Mathematical Software* (GAMS) recently developed by the National Institute of Standards and Technology (NIST). GAMS is both an on-line cross-index of available mathematical software as well as a repository of some 9,000 high-quality problem-solving modules from more than 80 software packages. It provides centralized access to such items as abstracts, documentation and source code of the software modules that it catalogues. Most of this software represents Fortran subprograms for mathematical problems which commonly occur in computational science and engineering, such as solution of systems of linear algebraic equations, computing matrix eigenvalues, solving nonlinear systems of differential equations, finding minima of nonlinear functions of several variables, evaluating the special functions of applied mathematics, and performing nonlinear regression. Among the packages catalogued in GAMS are:

- the IMSL, NAG, PORT, and SLATEC libraries;
- the BLAS, EISPACK, FISHPAK, FNLIB, FFTPACK, LAPACK, LINPACK and STARPAC packages;

- the DATAPLOT and SAS statistical analysis systems;
- the netlib routines, including the Collected Algorithms of the ACM (see below).

Note that although GAMS catalogues both public-domain and proprietary software, source code of proprietary software is not available through GAMS, although related items such as documentation and example programs often are. Software can be found either by browsing through a decision tree or performing a key-word search. GAMS can be accessed in several ways:

- telnet gams.nist.gov
- gopher gams.nist.gov
- <www browser> http://gams.nist.gov, where <www browser> is a World Wide Web browser such as Mosaic or netscape.

Report any questions or problems to gams@cam.nist.gov. For more details, login to the system or see (Boisvert, Howe and Kahaner 1985; Boisvert 1990).

Included in the software catalogued by GAMS are many high-quality public-domain routines available by electronic mail (e-mail) from *netlib*. These routines are now also available through Xnetlib, a more sophisticated X interface to netlib and the NA-Net Whitepages, or through the World Wide Web at the address http://www.netlib.org/index.html. For more information on netlib, see (Dongarra and Grosse 1987; Dongarra, Rowan and Wade 1995), send the message "send index" by e-mail to either netlib@ornl.gov or netlib@research.att.com, or access http://www.netlib.org/index.html through the World Wide Web.

The ACM Transactions on Mathematical Software publishes refereed public-domain software. These high-quality routines, covering a broad range of problem areas, are included in the Collected Algorithms of the ACM, available through both GAMS and netlib.

Not mentioned above are the commercial interactive packages MATLAB[2], Maple[3] and Mathematica[4]. MATLAB is built upon a foundation of sophisticated matrix software and includes routines for solving many standard mathematical and statistical problems. In addition, "toolboxes" for several application areas, such as control theory, are available. Both Maple and Mathematica are primarily symbolic algebra packages, but contain many high-quality numerical routines as well.

Glossary

An $m \times n$ matrix is **banded** if all its nonzero elements occur in a band around its main diagonal.

[2] For more information on MATLAB, contact The MathWorks Inc., 24 Prime Park Way, Natick, MA 01760; Phone: (508) 653–1415; Fax: (508) 653–2997; e-mail: info@mathworks.com.

[3] For more information on Maple, contact Waterloo Maple Software, 450 Phillip St., Waterloo, Ontario, Canada, N2L 5J2; Phone: (519) 747–2373; Fax: (519) 747–5284; e-mail: info@maplesoft.on.ca.

[4] For more information on Mathematica, contact Wolfram Research Inc., 100 Trade Center Dr., Champaign, IL 61820–7237; Phone: (217) 398–0700; Fax: (217) 398-0747; e-mail: info@wri.com.

Given a $n \times n$ matrix A, $\det(A - \lambda I) = 0$ is called the **characteristic equation** of A and the polynomial $p(\lambda) = \det(A - \lambda I)$ of degree n is called the **characteristic polynomial** of A.

A matrix A is **column diagonally dominant** if A^T is row-diagonally dominant.

The **complex-conjugate transpose** of an $m \times n$ matrix $A = [a_{ij}]$ is the $n \times m$ matrix $A^H = [a^h_{ij}]$ where $a^h_{ij} = \bar{a}_{ji}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. The complex-conjugate transpose of a column (row) n-vector $x = [x_i]$ is the row (column) vector $x^H = [x^h_i]$ with $x^h_i = \bar{x}_i$ for $i = 1, \ldots, n$.

An $m \times n$ matrix $D = [d_{ij}]$ is **diagonal** if $d_{ij} = 0$ for $i \neq j$. D is **block diagonal** if each d_{ij} in the definition above is a submatrix rather than a single number.

A matrix A is **diagonally dominant** if either A or A^T is row-diagonally dominant.

Given a $n \times n$ matrix A, the (possibly complex) number λ is called an **eigenvalue** of A and the nonzero vector x is called the associated **eigenvector** of A if $Ax = \lambda x$.

The **Euclidean norm** of a vector x is $\|x\|_2 = \sqrt{x^H x}$. The Euclidean norm is often called the 2-norm.

A **flop**, short for floating-point operation, is a multiplication and either an addition or a subtraction.

An $n \times n$ matrix A is **Hermitian** if $A = A^H$, where A^H is the complex-conjugate transpose of A.

An $n \times n$ matrix A is **Hermitian positive (negative) definite** if A is Hermitian and $z^H Az > 0$ ($z^H Az < 0$) for all complex n-vectors $z \neq 0$.

A matrix A is **Hessenberg** if it is either upper or lower Hessenberg. Note that a symmetric Hessenberg matrix is tridiagonal.

The $k \times k$ **leading principal minor** of a matrix A is the $k \times k$ submatrix in the top left corner of A.

A matrix A is **lower Hessenberg** if $a_{ij} = 0$ for $i < j - 1$. That is, it is lower triangular except for a single non-zero super-diagonal.

An $n \times n$ matrix $L = [l_{ij}]$ is **lower triangular** if $l_{ij} = 0$ for $1 \leq i < j \leq n$ and **strictly lower triangular** if $l_{ij} = 0$ for $1 \leq i \leq j \leq n$. L is **block lower triangular** or **block strictly lower triangular**, respectively, if each l_{ij} in the definitions above is a submatrix rather than a single number.

An $m \times n$ matrix Q is **orthogonal** if $Q^T Q = I$.

A **permutation** matrix is an $n \times n$ matrix with exactly one 1 in each row and column and all other elements equal to 0.

The **rank** of a matrix is the maximal number of independent rows (or columns) of the matrix.

An $m \times n$ matrix $R = [r_{ij}]$ is **right triangular** if $r_{ij} = 0$ for $i > j$. If $m = n$, the terms right triangular and upper triangular are equivalent.

An $m \times n$ matrix A with $m \leq n$ is **row diagonally dominant** if

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| < |a_{ii}| \qquad \text{for } i = 1, \ldots, m.$$

Two matrices A and B are **similar** if $B = WAW^{-1}$ for some non-singular matrix W. The matrix W is the associated **similarity transformation**.

An $m \times n$ matrix A is **sparse** if the number of nonzeros in A is much less than mn, the total number of elements in A.

The **spectral radius** of a square matrix A is $\rho(A) = \max\{|\lambda| : \lambda \text{ an eigenvalue of } A\}$.

For **strictly lower triangular**, see lower triangular.

For **strictly upper triangular**, see upper triangular.

An $n \times n$ matrix A is **symmetric** if $A = A^T$, where A^T is the transpose of A.

A real $n \times n$ matrix A is **symmetric indefinite** if A is symmetric and $x^T Ax > 0$ for some real n-vector x and $y^T Ay < 0$ for some real n-vector y.

A real $n \times n$ matrix A is **symmetric positive (negative) definite** if A is symmetric and $x^T Ax > 0$ ($x^T Ax < 0$) for all real n-vectors $x \neq 0$.

A real $n \times n$ matrix A is **symmetric positive (negative) semidefinite** if A is symmetric and $x^T Ax \geq 0$ ($x^T Ax \leq 0$) for all real n-vectors $x \neq 0$.

The **transpose** of an $m \times n$ matrix $A = [a_{ij}]$ is the $n \times m$ matrix $A^T = [a^t_{ij}]$ where $a^t_{ij} = a_{ji}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. The transpose of a column (row) n-vector $x = [x_i]$ is the row (column) vector $x^T = [x^t_i]$ with $x^t_i = x_i$ for $i = 1, \ldots, n$.

An $n \times n$ matrix $L = [l_{ij}]$ is **unit lower triangular** if L is lower triangular and $l_{ii} = 1$ for $i = 1, \ldots, n$.

A matrix A is **upper Hessenberg** if $a_{ij} = 0$ for $i > j + 1$. That is, it is upper triangular except for a single non-zero sub-diagonal.

An $n \times n$ matrix $U = [u_{ij}]$ is **upper triangular** if $u_{ij} = 0$ for $1 \leq j < i \leq n$ and **strictly upper triangular** if $u_{ij} = 0$ for $1 \leq j \leq i \leq n$. U is **block upper triangular** or **block strictly upper triangular**, respectively, if each u_{ij} in the definitions above is a submatrix rather than a single number.

An $m \times n$ matrix $A = [a_{ij}]$ is tridiagonal if $a_{ij} = 0$ for $|i - j| > 1$.

For **2-norm**, see Euclidean norm.

Mathematical Symbols Used

A^H: the complex-conjugate transpose of the matrix A.

A^T: the transpose of the matrix A.

$\mathbb{C}$: the set of complex numbers.

$\mathbb{C}^n$: the set of complex vectors with n components.

$\mathbb{C}^{m\times n}$: the set of complex $m \times n$ matrices.

$\mathcal{C}$: the set of continuous functions.

$\mathcal{C}[a,b]$: the set of continuous functions on the interval $[a,b]$.

$\mathcal{C}^p$: the set of continuous functions with p continuous derivatives.

$\mathcal{C}^p[a,b]$: the set of continuous functions with p continuous derivatives on the interval $[a,b]$.

$O(h^p)$ is any quantity that depends on h that can be bounded above by $C \cdot h^p$ for some constant C and all $h \in (0,H]$ for some $H > 0$.

$O(n^p)$ is any quantity that depends on n that can be bounded above by $C \cdot n^p$ for some constant C and all positive integers n.

$\mathbb{R}$: the set of real numbers.

$\mathbb{R}^n$: the set of real vectors with n components.

$\mathbb{R}^{m\times n}$: the set of real $m \times n$ matrices.

x^T: the transpose of the vector x.

$\|x\|$ and $\|A\|$ are norms of the vector x and the matrix A, respectively.

$\|x\|_2$ and $\|A\|_2$ are Euclidean norms (also called two-norms) of the vector x and the matrix A, respectively.

z^H: the complex-conjugate transpose of the vector z.

$\rho(A)$ is the spectral radius of a square matrix A.

Abbreviations Used

ACM: Association for Computing Machinery.

ADI: alternating direction implicit.

BC: boundary condition.

BVP: boundary-value problem.

CD: conjugate direction.

CG: conjugate gradient.

DOF: degrees of freedom.

FD: finite difference.

FDM: finite difference method.

FE: finite element.

FEM: finite element method.

GAMS: Guide to Available Mathematical Software.

GE: Gaussian elimination.

IC: initial condition.

ICF: incomplete Cholesky factorization.

IEEE: Institute of Electrical and Electronics Engineers.

IVP: initial-value problem.

LMF: linear multistep formula.

MM: multigrid method.

ODE: ordinary differential equation.

PCG: preconditioned conjugate gradient.

PDE: partial differential equation.

PP: piecewise polynomial.

RK: Runge-Kutta.

SD: steepest descent.

SOR: successive over relaxation.

SPD: symmetric positive-definite.

SSOR: symmetric successive over relaxation.

w.r.t.: with respect to.

References

Ames, W. F. 1992. *Numerical Methods for Partial Differential Equations.* Academic Press, New York.

Ascher, U. M., Mattheij, R. M. M., and Russell, R. D. 1988. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations.* Prentice-Hall, Englewood Cliffs, NJ.

Atkinson, K. E. 1989. *An Introduction to Numerical Analysis.* John Wiley & Sons, New York, second edition.

Axelsson, O. 1994. *Iterative Solution Methods.* Cambridge University Press, Cambridge.

Becker, E. B., Carey, G. F., and Oden, J. T. 1981. *Finite Elements*, volume I. Prentice Hall, Englewood Cliffs, NJ.

Bertsekas, D. P. and Tsitsiklis, J. N. 1989. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, NJ.

Boisvert, R. 1990. The guide to available mathematical software advisory system. In Houstis, E., Rice, J., and Vichnevetsky, R., editors, *Intelligent Mathematical Software Systems*, pages 167–178. North-Holland, Amsterdam.

Boisvert, R. F., Howe, S. E., and Kahaner, D. K. 1985. GAMS: A framework for the management of scientific software. *ACM Transactions on Mathematical Software*, 11(4):313–355.

Briggs, W. L. 1987. *A Multigrid Tutorial.* SIAM, Philadelphia.

Buchanan, J. L. and Turner, P. R. 1992. *Numerical Methods and Analysis.* McGraw-Hill, New York.

Butcher, J. C. 1987. *The Numerical Analysis of Ordinary Differential Equations.* John Wiley & Sons, New York.

Celia, M. A. and Gray, W. G. 1992. *Numerical Methods for Differential Equations.* Prentice Hall, Englewood Cliffs, NJ.

Christara, C. C. 1994. Quadratic spline collocation methods for elliptic partial differential equations. *BIT*, 34(1):33–61.

Conte, S. D. and de Boor, C. 1980. *Elementary Numerical Analysis.* McGraw-Hill, New York, third edition.

Cullum, J. and Willoughby, R. 1985. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations.* Birkhaüser, Boston. Vol. 1 Theory and Vol. 2 Programs.

Dahlquist, G. and Björck, Å. 1974. *Numerical Methods.* Prentice Hall, Englewood Cliffs, NJ.

Davis, P. J. 1975. *Interpolation and Approximation.* Dover, New York.

de Boor, C. 1978. *A Practical Guide to Splines.* Springer-Verlag, New York.

Dennis, J. E. and Schnabel, R. B. 1983. *Numerical Methods for Unconstrained Optimisation and Nonlinear Equations.* Prentice Hall, Englewood Cliffs, NJ.

Dongarra, J. and Grosse, E. 1987. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30(5):403–407.

Dongarra, J., Rowan, T., and Wade, R. 1995. Software distribution using XNETLIB. *ACM Transactions on Mathematical Software*, 21(1):79–88.

Duff, I., Erisman, A., and Reid, J. 1986. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford.

George, A. and Liu, J. 1981. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ.

Goldberg, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48.

Golub, G. H. and Van Loan, C. F. 1989. *Matrix Computations.* John Hopkins University Press, Baltimore, second edition.

Hackbusch, W. 1994. *Iterative Solution of Large Sparse Systems of Equations.* Springer-Verlag, New York.

Hageman, L. A. and Young, D. M. 1981. *Applied Iterative Methods.* Academic Press, New York.

Hager, W. W. 1988. *Applied Numerical Linear Algebra.* Prentice Hall, Englewood Cliffs, NJ.

Hairer, E., Nørsett, S. P., and Wanner, G. 1987. *Solving Ordinary Differential Equations I: Nonstiff Problems.* Springer-Verlag, New York.

Hairer, E. and Wanner, G. 1991. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems.* Springer-Verlag, New York.

Hall, C. A. and Porsching, T. A. 1990. *Numerical Analysis of Partial Differential Equations.* Prentice Hall, Englewood Cliffs, NJ.

Householder, A. 1970. *The Numerical Treatment of a Single Nonlinear Equation.* McGraw-Hill, New York.

Houstis, E. N., Christara, C. C., and Rice, J. R. 1988. Quadratic spline collocation methods for two-point boundary value problems. *Internat. J. Numer. Methods Engrg.*, 26:935–952.

IEEE 1985. *IEEE Standard for Binary Floating-Point Arithmetic.* American National Standards Institute, New York. ANSI/IEEE Std. 754–1985.

Johnson, L. W. and Riess, R. D. 1982. *Numerical Analysis.* Addison Wesley, Reading, Mass.

Kahaner, D., Moler, C., and Nash, S. 1989. *Numerical Methods and Software.* Prentice Hall, Englewood Cliffs, NJ.

Lambert, J. D. 1991. *Numerical Methods for Ordinary Differential Equations.* John Wiley & Sons, New York.

Ortega, J. M. 1988. *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, New York.

Parlett, B. N. 1968. Global convergence of the basic QR algorithm on Hessenberg matrices. *Math. Comput.*, 22:803–817.

Parlett, B. N. 1980. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, NJ.

Prenter, P. M. 1975. *Splines and Variational Methods.* John Wiley & Sons, New York.

Saad, Y. 1992. *Numerical Methods for Large Eigenvalue Problems.* Manchester University Press (John Wiley & Sons), New York.

Scott, D. 1981. The Lanczos algorithm. In Duff, I. S., editor, *Sparse Matrices and Their Uses*, pages 139–160. Academic Press, London.

Shampine, L. F. 1994. *Numerical Solution of Ordinary Differential Equations.* Chapman & Hall, New York.

Shampine, L. F. and Gear, C. W. 1979. A user's view of solving stiff ordinary differential equations. *SIAM Rev.*, 21:1–17.

Stoer, J. and Bulirsch, R. 1980. *Introduction to Numerical Analysis.* Springer-Verlag, New York.

Strang, G. and Fix, G. J. 1973. *An Analysis of the Finite Element Method.* Prentice Hall, Englewood Cliffs, NJ.

Strikwerda, J. C. 1989. *Finite Difference schemes and Partial Differential Equations.* Wadsworth and Brooks/Cole, Pacific Grove, Calif.

Van de Velde, E. F. 1994. *Concurrent Scientific Computing.* Springer-Verlag, New York.

Van Loan, C. F. 1992. *Computational Frameworks for the Fast Fourier Transform.* SIAM, Philadelphia.

Varga, R. S. 1962. *Matrix Iterative Analysis.* Prentice-Hall, Englewood Cliffs, NJ.

Wilkinson, J. H. 1965. *The Algebraic Eigenvalue Problem.* Oxford University Press, Oxford.

Young, D. M. 1971. *Iterative Solution of Large Linear Systems.* Academic Press, New York.

Further Reading

Ciarlet, P. G. 1989. *Introduction to Numerical Linear Algebra and Optimization.* Cambridge University Press, Cambridge.

Forsythe, G. E., Malcolm, M. A. and Moler, C. B. 1977. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, NJ.

Forsythe, G. E. and Moler, C. B. 1967. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, NJ.

Golub, G. H. and Ortega, J. M. 1992. *Scientific Computing and Differential Equations.* Academic Press, New York.

Golub, G. H. and Ortega, J. M. 1993. *Scientific Computing: An Introduction with Parallel Computing.* Academic Press, New York.

Isaacson, E. and Keller, H. B. 1966. *Analysis of Numerical Methods.* John Wiley & Sons, New York.

Press, W. H., Flannery, B. P., Teukolsky, S. A. and Vetterling, W. T. 1986. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, Cambridge.

Schultz, M. H. 1973. *Spline Analysis.* Prentice Hall, Englewood Cliffs, NJ.

Stewart, G. W. 1973. *Introduction to Matrix Computations.* Academic Press, New York.

Wilkinson, J. H. 1963. *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, NJ.

Tables

for $k = 1, \ldots, n-1$ do
    for $i = k+1, \ldots, n$ do
        $m_{ik} = a_{ik}^{(k-1)} / a_{kk}^{(k-1)}$
        for $j = k+1, \ldots, n$ do
            $a_{ij}^{(k)} = a_{ij}^{(k-1)} - m_{ik} \cdot a_{kj}^{(k-1)}$
        end
        $b_i^{(k)} = b_i^{(k-1)} - m_{ik} \cdot b_k^{(k-1)}$
    end
end

Table 1:

for $i = n, \ldots, 1$ do

$$x_i = \left(\tilde{b}_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii}$$

end

Table 2:

for $i = 1, \ldots, n$ do

$$\tilde{b}_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij} \tilde{b}_j \right) / l_{ii}$$

end

Table 3:

for $j = 1, \ldots, n$ do

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}$$

for $i = j+1, \ldots, n$ do

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right) / l_{jj}$$

end

end

Table 4:

	factor	solve	store A	store L
dense	$m^6/6$	m^4	$m^4/2$	$m^4/2$
banded	$m^4/2$	$2m^3$	m^3	m^3
sparse	$O(m^3)$	$O(m^2 \log m)$	$5m^2$	$O(m^2 \log m)$

Table 5:

choose an initial guess x_0
compute $r_0 = b - Ax_0$
solve $M\tilde{r}_0 = r_0$
set $p_0 = \tilde{r}_0$
for $k = 0, 1, \ldots$ until convergence do
$\alpha_k = r_k^T \tilde{r}_k / p_k^T A p_k$
$x_{k+1} = x_k + \alpha_k p_k$
$r_{k+1} = r_k - \alpha_k A p_k$
solve $M\tilde{r}_{k+1} = r_{k+1}$
$\beta_k = r_{k+1}^T \tilde{r}_{k+1} / r_k^T \tilde{r}_k$
$p_{k+1} = \tilde{r}_{k+1} + \beta_k p_k$
end

Table 6:

```
for j = 1, ..., n do                    (q_j and a_j are columns j
    q_j = a_j                            of Q and A, respectively)
    for i = 1, ..., j − 1 do
        r_ij = q_i^T q_j
        q_j = q_j − r_ij q_i
    end
    r_jj = ||q_j||_2
    q_j = q_j / r_jj
end
```

Table 7:

```
Pick z_0
for k = 1, 2, ... do
    w_k = A z_{k−1}
    Choose m ∈ {1, ..., n} such that
        |(w_k)_m)| ≥ |(w_k)_i)| for i = 1, ..., n
    z_k = w_k / (w_k)_m
    μ_k = (w_k)_m / (z_{k−1})_m
    test stopping criterion
end
```

Table 8:

```
Set A_0 = A
for k = 1, 2, ... do
    Compute the QR factorization of A_{k−1} = Q_k R_k
    Set A_k = R_k Q_k
    test stopping criterion
end
```

Table 9:

Pick z_0
for $k = 1, 2, \ldots$ do
    Solve $(A - \tilde{\lambda} I) w_k = z_{k-1}$
    $z_k = w_k / \|w_k\|_\infty$
    test stopping criterion
end

Table 10:

Pick an initial guess x_0 and a tolerance ϵ
for $k = 1, \ldots,$ maxit do
    $s_{k-1} = -\nabla f(x_{k-1})$
    if $\|s_{k-1}\| \leq \epsilon$ exit loop
    find $\alpha^* \in \mathbb{R}$ that minimizes $f(x_{k-1} + \alpha s_{k-1})$
    $x_k = x_{k-1} + \alpha^* s_{k-1}$
end

Table 11:

Pick an initial guess x_0 and a tolerance ϵ
Initialize $s_0 = 0$ and $\beta = 1$
for $k = 1, \ldots,$ maxit do
    if $\|\nabla f(x_{k-1})\| \leq \epsilon$ exit loop
    $s_k = -\nabla f(x_{k-1}) + \beta s_{k-1}$
    find $\alpha^* \in \mathbb{R}$ that minimizes $f(x_{k-1} + \alpha s_k)$
    $x_k = x_{k-1} + \alpha^* s_k$
    $\beta = \|\nabla f(x_k)\|^2 / \|\nabla f(x_{k-1})\|^2$
end

Table 12:

Quad. Rule	n	d	Interpolant	$Q(f)$	$I(f) - Q(f)$
rectangle	1	0	constant	$(b-a)f(a)$	$\frac{(b-a)^2}{2}f'(\eta)$
midpoint	1	1	constant	$(b-a)f(m)$	$\frac{(b-a)^3}{24}f''(\eta)$
trapezoidal	2	1	linear	$\frac{b-a}{2}[f(a)+f(b)]$	$-\frac{(b-a)^3}{12}f''(\eta)$
Simpson's	3	3	quadratic	$\frac{b-a}{6}[f(a)+4f(m)$ $+f(b)]$	$-\frac{(b-a)^5}{2880}f^{(4)}(\eta)$
corrected trap.	4	3	cubic	$\frac{b-a}{2}[f(a)+f(b)]$ $+\frac{(b-a)^2}{12}[f'(a)-f'(b)]$	$\frac{(b-a)^5}{720}f^{(4)}(\eta)$
Newton-Cotes	n	$\geq n-1$	deg. $n-1$	see §9.1.1	$\frac{(b-a)^{d+2}}{K}f^{(d+1)}(\eta)$
Gaussian	n	$2n-1$	deg. $n-1$	see §9.1.2	$\frac{(b-a)^{d+2}}{C}f^{(d+1)}(\eta)$

Table 13:

Quad. Rule	n	d	PP Interp.	Formula	Error
rectangle	s	0	constant	$h\sum_{i=0}^{s-1} f(a+ih)$	$\frac{h}{2}(b-a)f'(\eta)$
midpoint	s	1	constant	$h\sum_{i=1}^{s} f(a+(i-1/2)h)$	$\frac{h^2}{24}(b-a)f''(\eta)$
trapezoidal	$s+1$	1	linear	$\frac{h}{2}[f(a)+f(b) +2\sum_{i=1}^{s-1} f(a+ih)]$	$-\frac{h^2}{12}(b-a)f''(\eta)$
Simpson's	$2s+1$	3	quadratic	$\frac{h}{6}[f(a)+f(b) +2\sum_{i=1}^{s-1} f(a+ih) +4\sum_{i=1}^{s} f(a+(i-1/2)h)]$	$-\frac{h^4}{2880}(b-a)f^{(4)}(\eta)$
corrected trap.	$s+3$	3	cubic Hermite	$\frac{h}{2}\left[f(a)+f(b) +2\sum_{i=1}^{s-1} f(a+ih)\right] +\frac{h^2}{12}[f'(a)-f'(b)]$	$\frac{h^4}{720}(b-a)f^{(4)}(\eta)$

Table 14:

```
subroutine AQ(a, b, ϵ)
    (Q, E) = LQM(a, b)
    if (E ≤ ϵ) then
        return (Q, E)
    else
        m = (a+b)/2
        return AQ(a, m, ϵ/2) + AQ(m, b, ϵ/2)
    end
end
```

Table 15: